Code Generation from Pragmatics Annotated Coloured Petri Nets

Simonsen, Kent Inge

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Code Generation from Pragmatics Annotated Coloured Petri Nets

by

Kent Inge Fagerland Simonsen

Kongens Lyngby 2014
PHD-2014-345

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk PHD-2014-345

Summary (English)

All electronic communication relies on communication protocols. It is therefore very important that protocols are correct and that protocol implementations are reliable. Coloured Petri Nets (CPNs) have been widely used to model, analyse and verify communication protocols. However, relatively limited work has been done on transforming CPN model to protocol implementations. The goal of the thesis is to be able to automatically generate high-quality implementations of communication protocols based on CPN models.

In this thesis, we develop a methodology for generating implementations of protocols using a sub-class of CPNs, called Pragmatics Annotated CPNs (PA-CPNs). PA-CPNs give structure to the protocol models and allows the models to be annotated with code generation pragmatics. These pragmatics are used by our code generation approach to identify and execute the appropriate code generation templates. The templates hold the information needed to transform the model to a fully working protocol implementation for a target platform. The code generation approach coupled with PA-CPNs provide a flexible way to perform code generation for communication protocols. The code generation approach has been implemented in a prototype tool called PetriCode.

We defined several criteria for our code generation approach, the approach should be *scalable* so that is can be used to generate code for industrial sized protocols. The models should be *verifiable* and it should be possible to perform efficient verification on the models. The approach and the models that are employed for code generation should be *platform independent* in the sense that it should be possible to generate code for a wide range of platforms based on the same model. The generated code should be *integrable* meaning that it should

be able to use different third party libraries and the code should be easily usable by third party code. Finally, the code should be *readable* by developers with expertise on the considered platforms.

In this thesis, we show that our code generation approach is able to generate code for a wide range of platforms without altering the PA-CPN model that describe the protocol design. The generated code is also shown to be readable and we demonstrate that a generated implementation can be easily integrated with third party software. We also show that our approach scales to industrial sized protocols by applying our approach to generate code for the WebSocket protocol. The WebSocket protocol creates a message-based two-way channel that can be used by web applications. This allows web applications to communicate with the server much more efficiently than using the traditional request-response pattern for certain application types such as games and rich web applications. Finally, we conclude the evaluation of the criteria of our approach by using the WebSocket PA-CPN model to show that we are able to verify fairly large protocols.

Summary (Danish)

Alt elektronisk kommunikation bygger på kommunikasjonsprotokoller. Det er derfor viktigt at protokoller er korrekte og at implementationer af protokoller er pålidelige. Farvede Petri Nets (CPNs) har været bredt anvendt til modellering, analyse og verifikation af kommunikationsprotokoller, men der eksisterer relativt begrænsede forskningsresultater for at transformere CPN modeller til protokol implementationer. Målet med denne afhandling er at gøre det muligt automatisk at generere høj-kvalitets imlementationer af kommunikationsprotokoller baseret på CPN modeller.

I denne afhandling utvikles en metode for at generere implementationer af protokoller baseret på en underklasse av CPN modeller kaldet Pragmatic-Annoterede CPN (PA-CPN) modeller. PA-CPN modeller giver struktur til protokolmodeller og tillader modellerne at blive annoteret med kodegenereringspragmatikker. Pragmatikkerne bliver brukt til at identificere og utføre kodegenereringsskabeloner. Skabelonerne inndeholder den information som skal til for at transformere en model til en komplet kørende implementation for en given platform. Tilgangen til kode generering koblet med PA-CPN modeller udgør en fleksibel måde at generere kode for kommunikationsprotokoller. Kodegenereringsmetoden er implementeret i en prototype værktøj kaldet PetriCode.

Vi opstiller flere kriterier for tilgangen til kodegenerering. Den skal være skalerbar for at den kan brukes til at generere kode for protokoller af industriel størrelse. Modellerne skal være verificerbare og det skal være muligt at utføre verifikationen på modellerne effektivt. Tilgangen og de modeller som anvendes til kodegenerering skal være platform uafhængige hvilket betyder at det skal være muligt at gererere kode for et bred spektrum af platforme baseret på den

samme model. Den genererede kode skal være integrerbar så den kan bruges af trejdeparts biblioteker og koden skal let kunne brukes af tredjeparts kode. Endelig skal den genererede kode være læsbar af utviklere med ekspertise på de platforme som der genereres kode til.

I afhandlingen viser vi hvordan vores tilgang til kode generering kan generere kode for et bredt spektrum af platforme uden at ændre i den PA-CPN model som beskriver protokol designet. Vi viser at den genererede kode er læsbar og at den let kan integreres i trejdeparts software. Vi viser også at vores tilgang skalerer til protokoller af industriel størrelse ved at anvende den til at generere kode for WebSocket protokollen. WebSocket protokollen giver en beskedorienteret tovejskanal som kan bruges av web-applikationer. Dette tillader web-applikationer at kommunikere med en server mere effektivt end ved at bruge det traditionelle request-respons mønster for applikationer som eksempelvis spil og klient-tunge web applikationer. Vi afslutter evalueringen af kriterier for metoden ved at bruge PA-CPN modellen af WebSocket protokollen til at demonstrere at vi kan verificere komplekse protokoller.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring a Ph.D. in Informatics. The author has been employed as a Ph.D. research fellow at the Department of Computing, Mathematics, and Physics at Bergen University College supported by the strategic research program on Software Technologies for Distributed Systems (DISTECH).

The thesis deals with automatically generating implementations of communication protocols. This fills a gap in the ecosystem around the Coloured Petri Nets modelling language that is supported by the CPN Tools modelling tool.

The thesis consists of a collection of several papers and an overview article. All but one of the papers have been published at peer reviewed conferences and workshops. The overview article gives an overview of the results contained in the papers and puts it into a coherent context via a discussion of related work found in the literature.

Bergen, 30-August-2014

Kent Inge Fagerland Simonsen

Acknowledgements

I would not have been able to write this thesis without the help and support from many people and institutions. First of all, I would like to thank my supervisors Lars Michael Kristensen and Ekkart Kindler for their invaluable support and guidance throughout this project. I am grateful for all the time and effort they have devoted to this project. The input and advice they have given me through more or less regular meetings and less regular conference calls have been crucial to completing this thesis. Also, the seemingly endless stream of comments and corrections on the text of this theses and the attached papers have elevated the quality of the work significantly.

I would also like to thank the Bergen University College (BUC) and the department of computing, mathematics and physics for allowing me to embark on this project and paying my bills. Also, I want to thank the Technical University of Denmark (DTU) and the section for software engineering at DTU Compute for allowing me, as a non-employee of DTU, to enter its Ph.D. program.

The staff at BUC and DTU is also to be thanked for making me feel welcome at both institutions. And a special thanks should be extended to Sven-Olai Høyland for all the help with teaching Android and Java programming. A special thanks also goes out to Adrian Rutle, who encouraged me to apply for the PhD research fellow position at BUC, even though I applied for the wrong position.

I also thank my fellow Ph.D. students and other colleges at BUC and DTU who I have shared many a laugh with. Especially I want to that Florian Mantz and Piotr Henryk Kazmierczak whom I have also shared office with from the very

start of my PhD studies.

Finally, I want to thank my extraordinarily patient wife Namfon Phouthonesy for her support and encouragement without which this thesis may never have been finished. I also want to thank my parents for encouragement and support in these years. Finally I wish to thank all my friends for putting up with me talking about my thesis these years.

Contents

Part I

Overview

Chapter 1

Introduction

In this thesis we present an approach for automatically generating implementations of communication protocols based on formal models.

1.1 Model Driven Software Engineering

Model-driven software engineering (MDSE) [BCW12] is a paradigm of software development where models are regarded as first-class entities of the development process. Models are often used to automatically generate (parts of) software systems by means of model-to-model (M2M) [JABK08] and model-to-text (M2T) [Obj08] transformations. This is in contrast to traditional software development methods where source code is the primary artefact of the development process while models, if they are used, are mainly used for design and documentation purposes. Throughout this thesis, we use the term MDSE to describe acts of creating software using models in a broad sense.

Models are used to describe complex systems in simple terms. Models, particularly the models that are used in MDSE, often abstract away details of a system under consideration to make the model easier to comprehend. In this thesis we present models for networking protocols that we then translate into implementations of the protocols under consideration. We will mainly deal with

graphical models although models, in general, do not need a graphical representation. Restricting the models to graphical representations may force the models to be abstract since graphical models tend to be more verbose in terms of screen real-estate than corresponding textual models.

In MDSE, models are often used to model either structural or behavioural aspects of a software system. Structural, which are the most common model type [Pet13], describe the static structure of a system. The most common of these types of models are UML [Grob] class diagrams. Behavioural models describe the behaviour and dynamic aspects of a system. UML state charts and Petri Nets [Pet62, Rei85] are popular languages for behavioural models. In this thesis, we use the behavioural modelling language Coloured Petri Nets (CPNs). However, we use it to also model certain structural features.

1.2 Code Generation

Code generation [BCW12] is an integral part of MDSE approaches. Code generation has been used in software engineering since the first compilers were created for generating machine code from high-level programming languages.

Model-to-Text transformations (M2T) differ in that the starting point are models, like UML class diagrams [Grob], and the produced artefacts are textual programs in some high-level programming language, such as Java [GJSB05]. The text can then be compiled and run as a program written in a classical high-level programming language.

Model-to-Text (M2T) transformations are usually facilitated by some form of templates. There exists many template technologies designed for M2T transformations such as Java Emitter Templates (JET) [C+03, Pow04]. However, generic template languages can also be used.

Two categories of code generation are partial and full code generation. Partial code generation means that parts of the code are generated, but more code must be added in order to obtain a complete implementation. Partial code generation is commonly used together with structural models where static parts of the code can be generated, but the model does not contain sufficient information about the behaviour of the modelled system. Full code generation means that all code is generated and there is no need to manually alter the code. This approach is used when models contain both structural and behavioural information or the behavioural aspects are standardised to create, for example, editors. The code generation approach that is presented in this thesis is a full code generation

approach which means that we generate fully working software.

1.3 Communication Protocols

We have chosen to focus on code generation for the domain of communication protocols since it plays an important role in most IT systems. A prominent example is the vast amount of web applications that are in use today for, e.g., on-line banking, shopping, government administration, and entertainment. The services provided by these applications all rely on the protocols governing the operation of the Internet such as the Internet Protocol (IP) [DH98], the Transmission Control Protocol (TCP) [Pos81] and the Hypertext Transfer Protocol (HTTP) [FGM+99]. In addition to being an important domain, another reason to chose the communication protocol domain is that there exists extensive works on modelling and analysing protocols in the literature. This allows us to build upon previous works with regards to modelling and analysis.

The IETF WebSocket (WS) protocol [FM11] creates a bi-directional message-based channel based on the HTTP protocol. The WS protocol allows web applications to efficiently communicate with a server. WS is popular with web applications that require low latency efficient communication such as games and other rich web applications.

Other examples of protocols are telecommunication systems, logistic systems with sensors and actuators, and control systems in vehicles. All these systems rely heavily on communication and synchronisation between concurrently executing software components and subsystems. As protocols are required to support complex services that are critical to both the operation of companies and the everyday life of citizens, it is important that they are working correctly already from the initial deployment.

The specification of the protocol service and the protocol design is, in many cases, based on natural language descriptions. One example of this is the Request for Comments (RFC) [Cro69] documents published by the Internet Engineering Task Force (IETF) [I]. Natural language specifications of protocols often have many issues that need to be resolved before a properly working implementation can be obtained. One class of issues originates from the fact that such specifications are inherently ambiguous making it difficult to achieve interoperability between independent implementations. Another source of issues to resolve is that the specifications are often incomplete in that the behaviour of the protocol is not described for all cases. This motivates the use of formal modelling in order to remove ambiguities of natural language descriptions and

to provide a basis for automatically generated implementations.

1.4 Coloured Petri Nets

Petri Net [Pet62, Rei85] is a mathematical modelling language that is often used to model distributed systems and communication protocols in particular. Traditional, low level, Petri Nets such as Place/Transition nets have four types of elements: *places*, *transitions*, *arcs* and *tokens*.

In Petri Nets, transitions can fire by removing a token from an incoming place (or places) and creating a token on an outgoing place (or places). Petri Nets are inherently concurrent which means that several independent transitions firings can be seen as firing in parallel. Furthermore, Petri Nets have non-deterministic properties which means that all orderings of events can be tested by exploring all possibilities of non-deterministic choices. The concurrency capabilities of Petri Nets makes them well suited for modelling protocols where concurrency and non-determinism are inherent to the domain. Concurrency is inherent to the protocol domain since protocols consists of agents running on different hosts communicating over channels. The channels are non-deterministic in that network channels are often not reliable in terms of guaranteed delivery and the order or the time it takes for a message to reach its destination.

Coloured Petri Nets (CPNs) belong to the family of High-level Petri Nets and combine Petri Nets with the Standard ML (SML) programming language [Ull98]. Petri Nets provide the foundation of the graphical notation and the semantic foundation for modelling concurrency, synchronisation, and communication. The functional programming language SML provides primitives for representing sequential aspects of protocols (such as data manipulation and data types) and for creating compact and parameterisable models. Formal modelling and validation with CPNs is supported by CPN Tools [RWL+03] which provides support for construction, simulation, functional and simulation-based performance analysis of CPN models. The addition of data types and a high-level programming language offered by CPN (in contrast to ordinary Petri nets) is highly important when constructing Petri net models of protocols.

The advantage of CPNs (and formal description techniques in general) is that they are based on the construction of executable models that make it possible to observe and experiment with the protocol design prior to implementation and deployment e.g., using simulations. This typically leads to more complete protocol specifications since the model will not be fully operational until all parts of the protocol have been (at least abstractly) specified. Furthermore,

the construction of a formal and executable model helps to identify and resolve ambiguities that may be present in a natural language specification. Another advantage is the support for model abstractions that makes it possible to specify the operation of the protocol without being concerned with implementation details such as message layout. A model also makes it possible to explore larger usage scenarios of a protocol system than what is in many cases practically possible without models.

Although CPNs have been widely used to model, analyse and verify communication protocols, relatively limited work has been done on transforming CPN model to protocol implementations. In this thesis we present an approach to automatically generate high-quality implementations of communication protocols based on CPN models.

1.4.1 State Spaces and Verification

Verification of behavioural properties of protocols with CPNs [Kri10] is supported by explicit *state space exploration* [BK08a]. In its simplest form this approach involves computing a directed graph where the nodes corresponds to the set of reachable states of the CPN model and the arcs represent occurrences of transitions causing state changes. State spaces can be constructed fully automatically by the state space tool in CPN Tools and guarantees complete coverage of all executions. Model checking by state space exploration hence provides a highly systematic error-detection technique that make it possible to automatically (i.e., algorithmically) check whether a protocol has a formally stated desired property. In addition, state space methods have the advantage that counter examples (error-traces) can be automatically synthesised if the protocol does not satisfy a given property.

The main disadvantage of state space exploration is the inherent state explosion problem [Val98], and a multitude of advanced state space methods have been developed aimed at alleviating the inherent state explosion problem. Early work on addressing state explosion in the context of CPNs concentrated on computer tool support for, and initial experiments with, the equivalence [Jen94], symmetry [CEFJ96], and the stubborn set methods [Val91]. The symmetry and equivalence methods rely on constructing a condensed state space where each node represents an equivalence class of states and each arc represents an equivalence class of events. The symmetry method has, e.g., been applied on a mutual exclusion protocol [JK99] and an embedded systems protocol [LK00] which shows that is has practical applicability while the equivalence method has only been used on a small stop-and-wait protocol [JK03] due to the obligation of providing a manual soundness proof for the user-provided equivalence

relation. The stubborn set method relies on analysing enabling and disabling dependencies between transitions and use this to explore only a subset of the events in each state encountered during state space exploration. The rich SML-based inscription language which is a fundamental building block of the CPN modelling language, however, poses problems for the analysis of transition dependencies in the context of CPNs [KV98] – unless relying on an unfolding of the CPN model to the equivalent Place/Transition net. Hence, restrictions on the modelling language are required to apply the stubborn set method without relying on unfolding. The stubborn set method supports verification of a wide array of properties depending on the sets. Another widely used verification approach in the context of CPNs is based on the methodology of [BGH04a]. A central component of this approach is an explicit modelling of both the protocol and its service, and the use of finite-state automata language comparison as a criterion for checking that the protocol conforms to the specified service. Work on addressing the state explosion problem in the context of CPNs has generally concentrated on making more economical use of memory resources when exploring the state space. Memory is (in many cases) the limiting factor in state space exploration of CPN models due to the large state vectors. Recent work in this area resulted in the development of the sweep-line method [CKM01, JKM12]. The sweep-line suite of methods is aimed at on-the-fly verification, as opposed to first generating the complete state space and subsequently querying it, and exploits a notion of progress found in many distributed systems. Exploiting progress allows for the deletion of states from memory during a progress-first traversal of the state space. This in turn reduces peak memory usage. The sweep-line method has been used [GKB02, VABG08, GOBK04, GHB05] for the verification of several industrial-sized protocols specified using the CPN modelling language.

1.5 Goals and Contributions of this Thesis

This thesis presents an approach for automatically generating implementations of communication protocols based on formal models. The approach consists of a modelling methodology and a code generation technique with an accompanying tool to transform models into working implementations through code generation.

We have aimed to develop our code generation approach in an extensible and flexible manner in order to ensure that it is applicable to a large part of the protocol domain. In order to ensure that our approach would generate high quality code, we initially formulated a set of goals for our approach stated below. We also use these goals to evaluate our approach.

- **Readability.** The generated code should be readable in order to facilitate inspection. In particular, this will facilitate maintainability of the generated code and resources used for code generation by inspection of the generated code. Inspection of the generated code may also increase confidence in the correctness of the generated code.
- **Platform Independence.** It should be possible to use our approach to generate implementations on many platforms from a single model.
- **Integrability.** It should be possible to create software that uses the generated code (upwards integrability) and also possible for generated software to use a broad spectrum of software libraries (downwards integrability).
- **Verifiability.** Models are widely used to verify the correctness of communication protocols. Our modelling methodology should result in verifiable models. This will allow us to gain confidence that our models are correct and also verify the protocols under consideration.
- **Scalability.** Our approach should be applicable to protocols ranging from simple examples to large industrial protocols. This allows our code generation approach to be used in realistic projects.

The main body of work for this thesis has been presented in seven papers [KS13, SK12, SKK13a, SKK14, Sim14b, SK14b, Sim14a]. Six of the papers have been presented at peer-reviewed international workshops and conferences and one paper [SKK14] is planned to be published at a later time. In the following, we briefly present the main contributions of each paper.

In [KS13] we provide an overview of how the Coloured Petri Nets (CPNs) modelling language has been used to model and verify protocols prior to this project. The paper surveys, in detail, four projects where CPNs were used for modelling and verifying protocols. Furthermore, the paper briefly presents several other projects where CPNs have been applied to protocols. We used the insights gained through this survey to develop our modelling approach and methodology. This survey was also helpful in developing the concept of descriptive models that we proposed in [SK12].

In [SK12] we propose a modelling approach that creates descriptive models, i.e., models that are primarily used for describing a protocol but can also be used as a basis for creating models for code generation and verification. This paper also presents a model and verification of the WebSocket protocol which, to the best of our knowledge, had not been done before. The descriptive WebSocket model described in [SK12] is an early effort to create a modelling methodology for code generation which is refined and formally defined in later papers. We also use the WebSocket protocol to evaluate our code generation approach in [SK14b].

In [SKK13a] we present our code generation approach. First, this paper introduces a class of CPNs that is inspired by the descriptive models presented in [SK12]. This class of CPNs, that we call Pragmatics Annotated CPNs (PA-CPNs), can be used for code generation while preserving verifiability and descriptiveness. PA-CPNs were developed based on our experiences gained through the survey described in [KS13], and the descriptive WebSocket model described in [SK12]. After presenting PA-CPNs, the paper introduces the techniques used to generate code based on a PA-CPN model.

In [SKK14] we formally define PA-CPNs. Furthermore, in this paper, we show how PA-CPNs can be used for space-efficient verification using the sweep-line method for state space exploration. The formal definition of PA-CPNs provides an unambiguous description of PA-CPNs which allows others to create models that conform to the PA-CPN definition.

In [Sim14b] we present the PetriCode tool that implements the approach we introduced in [SKK13a]. The paper also shows how the tool can be applied to generate an implementation of a simple framing protocol. The PetriCode tool uses PA-CPN models as input and emits implementations of the modelled protocols. We use PetriCode to evaluate our approach in [SK14b] and [Sim14a] with respect to the requirements listed above.

In [SK14b] we evaluate scalability and verifiability by using our approach and the PetriCode tool to generate an implementation of the WebSocket protocol based on a PA-CPN model. We also show that this PA-CPN model is amenable to verification by verifying some simple termination properties through state space exploration.

In [Sim14a], we present an evaluation of platform independence, integrability and code readability by evaluating the code generated by PetriCode. In this paper, we evaluate our code generation approach by creating a PA-CPN model of a simple framing protocol and testing the capabilities of the PetriCode tool and the generated code.

1.6 Outline of the Dissertation

Part II of this thesis consists of the papers [KS13, SK12, SKK13a, SKK14, Sim14b, SK14b, Sim14a] that make up the main body of work in this thesis. The rest of Part I contains an overview of the papers in Part II and a coherent discussion of related works. The overview in Part I is organised as follows.

Chapter 2 discusses the use of CPNs in modelling and verification of communication protocols. The chapter presents a descriptive CPN model of the WebSocket protocol and summarises several related protocol modelling efforts. The chapter summarises the papers *Applications of Coloured Petri Nets for Functional Validation of Protocol Designs* [KS13] and *Towards a CPN-based Modelling Approach for Reconciling Verification and Implementation of Protocol Models* [SK12].

Chapter 3 describes the code generation approach developed in this project including the concept of pragmatics and the PA-CPN net class we have developed to make it possible to perform code generation from CPNs. Furthermore, we discuss related code generation techniques based on Petri Nets and related modelling languages. The chapter provides an overview of the paper *Code Generation From Pragmatics Annotated Coloured Petri Nets* [SKK13a]. This chapter also summarises part of the paper *A Formal Definition of Pragmatic Annotated Coloured Petri Nets for Automated Protocol Software Generation and Verification* [SKK14] which gives a formal definition of the PA-CPN net class.

Chapter 4 describes the code generation tool, PetriCode, that implements the code generation approach described in chapter 3 of this thesis. This chapter also discusses technologies related to the implementation of PetriCode. This chapter summarises the paper *PetriCode: A Tool for Template-based Code Generation from CPN Models* [Sim14b].

Chapter 5 discusses the evaluation of the code generation approach and the PetriCode tool. The chapter gives an overview of the papers *An Evaluation of Automated Code Generation with the PetriCode Approach* [Sim14a] and *Implementing the Web Socket Protocol based on Formal Modelling and Automated Code Generation* [SK14b].

Chapter 6 presents the conclusions of this thesis and outlines directions for future work.

The reader is assumed to have a working knowledge of general software engineering and basic familiarity with modelling and Petri Nets. Part I of this thesis is intended to be read in the order it is written. However, readers with knowledge of CPNs and how they are used to model protocols can skip Chap. 2 as it mainly deals with CPNs and protocol modelling. Readers that are mainly interested in the conceptual aspects of this thesis could also skip Chap. 4 and

Chap. 5 as they mainly deal with the PetriCode tool and the evaluation of our approach.

In the course of the work done for this thesis project some other papers, technical reports and extended abstracts that are not included in this thesis have been published. These publications are listed below.

- [SMR10] K.I.F. Simonsen, A. Mantz, F. Rossini, and A. Rutle. Groovy and Grails meets Eclipse Modelling Framework. *Norsk informatikkonferanse (NIK)*, pages 34–43, 2010.
- [Sim11] K.I.F. Simonsen. On the use of Pragmatics for Model-based Development of Protocol Software. In *Proc of PNSE '11*, volume 723 of *CEUR Workshop Proceedings*, pages 179–190. CEUR-WS.org, 2011.
- [SKK12] K.I.F. Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocols from CPN models Annotated with Pragmatics – Extended Abstract. In *Proc of* 24^{th} *Nordic Workshop on Programming Theory*. NWPT, November 2012.
- [SKK13b] K.I.F Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocol Software from CPN models Annotated with Pragmatics. Technical Report IMM-Technical Reports-2013-01, Technical University of Denmark, DTU Informatics, January 2013. Available via `http://bit.ly/WwH2hf`.
- [KS14] S.A. Kumar and K.I.F. Simonsen. Towards a Model-Based Development Approach for Wireless Sensor-Actuator Network Protocols. In *Proc. of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, pages 35–39. ACM, 2014.
- [SK14a] K.I.F. Simonsen and L.M. Kristensen. A Pragmatic Approach for Transforming Coloured Petri Net Models Into Code: A Case Study of the IETF WebSocket Protocol. In *Abstracts of the 1st International Joint Symposium on Program and Model Transformations*, 2014.

The PetriCode tool along with examples is available from the `http://www.petricode.org`.

Chapter 2

Coloured Petri Nets for Modelling and Verification of Protocols

This chapter introduces CPNs and shows how they can be used for modelling and verifying in a protocol engineering context. First, CPNs are introduced using the WebSocket (WS) protocol as an example. Next, this chapter discusses related work in terms of other significant uses of CPNs for modelling and verification of protocols.

This chapter is based on the following papers:

- [SK12] K.I.F. Simonsen and L.M. Kristensen. Towards a CPN-based Modelling Approach for Reconciling Verification and Implementation of Protocol Models. In *Proc. of MOMPES'12*, volume 7706 of *LNCS*, pages 106–125. Springer, 2012.

- [KS13] L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *ToPNoc VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.

2.1 CPN Modelling of the WebSocket Protocol

This section introduces CPNs using a descriptive specification model of the WS protocol. A descriptive model is a model that has, as its primary purpose, to describe the system under consideration to the reader. This example demonstrates the most relevant CPN concepts for the code generation approach presented in subsequent chapters of this thesis as well as the WS protocol that will also be used as an example throughout Part I of this thesis. We do not present the entire model here and refer the reader to [SK12] for further details.

The WS protocol [FM11], which is developed by the IETF, provides a message-oriented connection between a client and a server on top of the Transmission Control Protocol (TCP). The WS protocol targets web applications and uses the Hypertext Transfer Protocol (HTTP) to open connections. For data transfer, the WS protocol relies directly on bi-directional TCP streams in order to avoid the request-response (polling) pattern of HTTP, and to eliminate the overhead induced by the verbose HTTP headers. Data framing is used on top of the TCP streams to make the WS connections message-oriented.

Figure 2.1 shows the top-level module of the CPN model which describes the overall architecture of the WS protocol. The WS protocol has two principal actors: one client and one server which are represented by the *substitution transitions* Client and Server, respectively. Substitution transitions represent underlying modules and are drawn as rectangles with double-lined borders. The name of the sub-module the substitution transition is connected to is shown in a smaller rectangular tag below the substitution transition and will be discussed later.

The two principal actors are connected by *places*, drawn as ellipses, representing channels from ClientToServer and from ServerToClient. Below the places the *colour set* of each of the places are written. The colour set (type) defines the

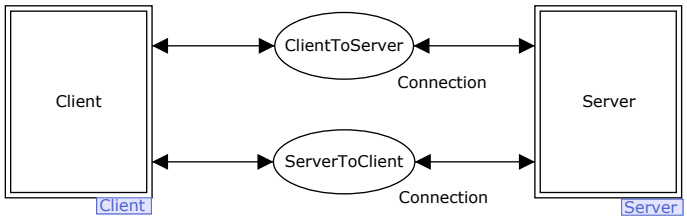


Figure 2.1: The top-level module of the descriptive specification model

```
colset Channel = with TCP;
colset Content = list Data;
colset Connection = product Channel * Content;
```

Figure 2.2: Colour set definition for channels and connections

colours, or data values, of the tokens that can be present at the place. Colour sets are defined using the type system of SML. Both places have the colour set `Connection` that is being used to model the TCP connection on top of which the WS connection is being established. The colour set `Connection` determines the kind of *tokens* that can reside on the two places modelling communication channels and are defined in Fig. 2.2. The `Connection` colour set is a product type where the first component (`Channel`) specifies the type of the channel to be used for communication, and the second component (`Content`) is used to model the data currently in transmission on the channel. The `Channel` type is defined to contain a single element: `TCP`. The `Data` type is defined as a union of all the types that are sent over the channel and is left out of Fig. 2.2 in order to promote the readability of this section. In general, the state of the CPN model (called a *marking*) consists of the distribution of tokens on the places of the CPN model.

Figure 2.3 shows the sub-module of the Client substitution transition from Fig. 2.1 The main states of the client principal in the WS protocol is modelled as places. In the initial state, READY, the client has not yet had any interaction with the server. The client is in the READY state when there is a token on the place READY. Once the WS connection has been established (the module in the substitution transitions EstablishWebSocketConnection has been completed), the principals enter the OPEN state by adding a token to the OPEN place and removing it from READY. In the OPEN state, data transfer (substitution transition DataTransfer) can take place, until either of principals chooses to close the connection (substitution transition CloseWebSocketConnection). After the WS connection has been closed, the principals enter a CLOSED state, and no further communication is possible.

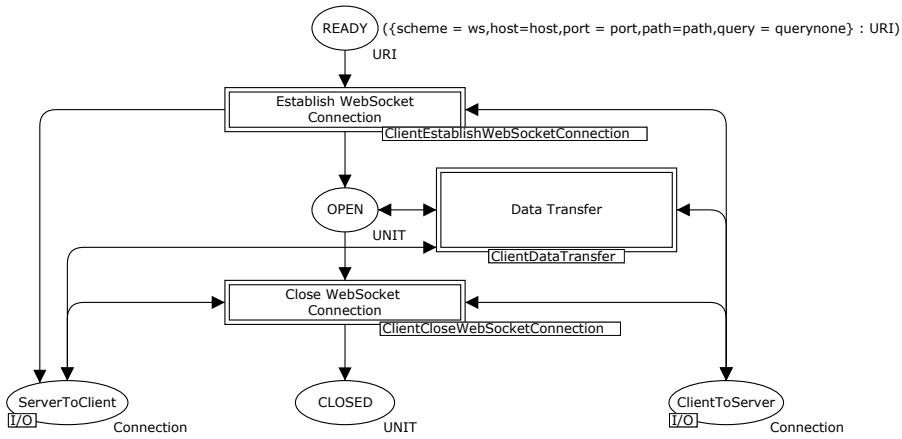


Figure 2.3: The top level module of the client principal

2.1.1 Opening the WebSocket Connection

The establishment of a WS connection is referred to as an *open handshake*. The client side of the open handshake module is shown in Fig. 2.4 and is associated with the substitution transition EstablishWebSocketConnection in Fig. 2.3. The first step is to open a TCP connection. The TCP connection is modelled as being open when there is a token present on the channel place. Opening the TCP connection (in the client to server direction) is modelled by the OpenTCP-Connection transition by placing a token on the channel place ServerToClient. The ServerToClient place is a *port place*, which means that it is connected to a *socket place* in an upper layer module. In this case, the place is connected to the socket place ServerToClient in Fig. 2.3 which is again a port place and is associated with the socket place ServerToClient in Fig 2.1.

A transition is said to be *enabled* when it is possible to compute a *binding*, a mapping of all variables needed by a transition to values based on the current marking, that fulfils all requirements imposed by arc inscriptions and the transition *guard* (if present). The OpenTCPConnection transition is enabled when there is one or more tokens on the READY place since the *variable* uri expects a token of the URI colour set and there are no further arc inscriptions or guards that must be fulfilled for this transition. When the OpenTCPConnection *occurs* (fires) the token at the READY place is removed and a copy is placed at the OPENING place. In addition, a token representing an open channel is placed at the ServerToClient place. Since the READY place has one token in the initial marking and no incoming arcs on any module level (see also Fig. 2.3), the

OpenTCPConnection will not become enabled again later in the execution of the model.

Figure 2.4 shows the ClientEstablishWebSocketConnection module that opens a WebSocket connection for the client. When the TCP connection is opened in the OpenTCPConnection transition, the client sends a HTTP upgrade request in the SendOpeningHandshake transition. The request is created by the function which creates a HTTP request shown in Fig. 2.5 called `ClientOpenHandshake`. The `method` field of a HTTPREQ message specifies that an HTTP GET operation is to be performed and the `uri` is used to identify the endpoint of the WS connection in the `host` and `resource` fields. The `upgrade` and `connection` fields indicate that this is an upgrade request for a WS connection. The `secwebsocketkey` field contains a base-64 encoded 16-byte nonce, but this type is in the model abstracted to a unit (`()`) value belonging to the colour set `UNIT` which contains a single value `()` denoted unit. The unit type in CPNs are similar to the black tokens of low-level Petri Nets. The `secwebsocketversion` indicates the version of the web socket protocol to be used.

After the client receives and validates the response from the server (modelled by the `ValidateOpeningHandshake` transition in Fig. 2.4), the client enters the OPEN state. At this point both the client and the server are ready to send and receive frames and the open handshake is finished.

2.1.2 Data Transfer

Once both the client and server are in the OPEN state they may transmit data until they send or receive a close frame. In addition, they may send ping and pong frames (to check that the connection is still alive). The module modelling the data transfer phase is shown in Fig. 2.6. Since the WS connection is bidirectional, the sending and receiving of data are independent operations. This is reflected by modelling sending and receiving as separate sub-modules as represented by the ClientSendMessage and ClientReceiveMessage substitution transitions of the data transfer module. The ClientPingPong substitution transition contains the modules for sending and receiving ping and pong messages.

The sending process for a message is shown in Fig. 2.7. When the client wishes to send a message, this is done by sending a sequence of frames. The transition SendNextDataFrame sends each non-final fragment by concatenating a fragment to a list of fragments that is a part of the token, of colour Connection, on the ClientToServer place. When the final frame has been sent, by firing the SendFinalDataFrame transition, the client is Ready to send the next message.

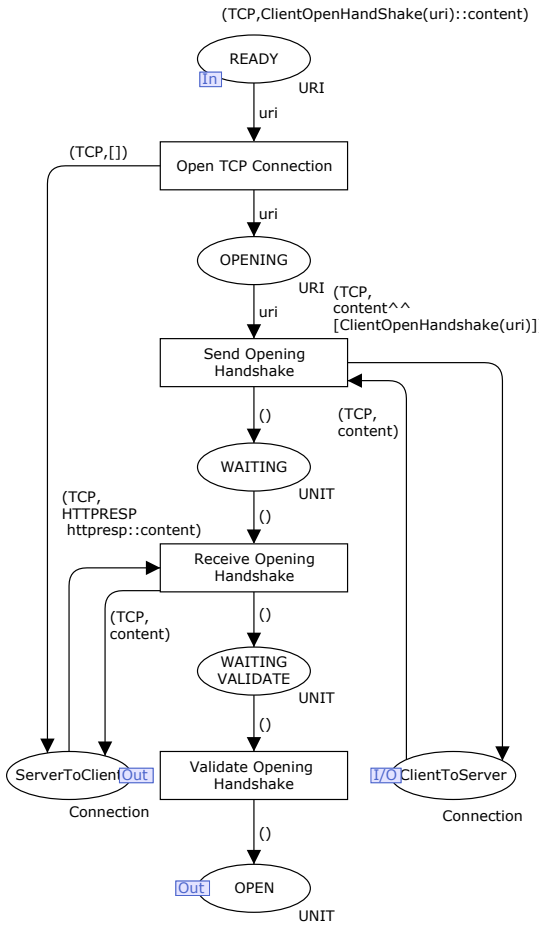


Figure 2.4: The WebSocket open handshake - client side

2.1.3 Closing the WebSocket Connection

In the OPEN state, either principal may initiate the closing of the WS connection by sending a close frame. The closing process of the client side is modelled by the module shown in Fig. 2.8 in the case where the server initiates closing. Upon receiving a close frame, the client removes the token from the OPEN place indicating that the protocol is now in the CLOSING state. In the closing state, the client sends a close frame to the server and enters the CLOSED state. When both the server and the client are in the CLOSED state, the WS protocol is said to be completed and the underlying TCP connection is closed.

```
fun ClientOpenHandshake (uri:URI) =
 HTTPREQ {
   method  = GET,          resource   = #path uri,
   version = HTTP_VERSION, host       = #host uri,
   upgrade = UPGRADE,      connection = CONNECTION,
   secwebsocketkey     = (),
   secwebsocketversion = WEBSOCKETVERSION
  };
```

Figure 2.5: Client open handshake function

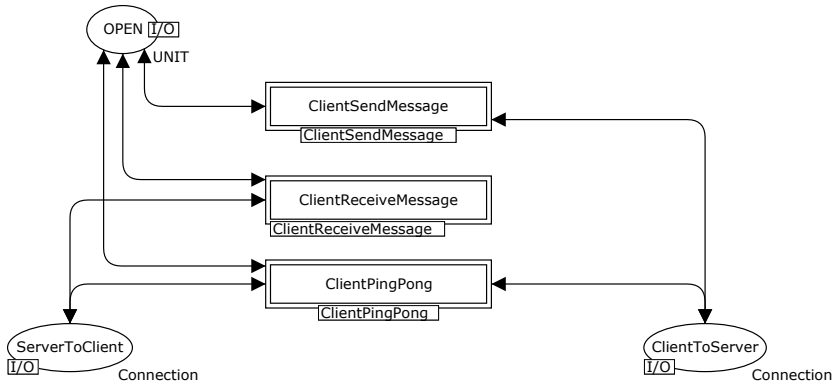


Figure 2.6: The data transfer phase - client side

2.2 Verification of the WebSocket Model

The purpose of the model presented in Sect. 2.1 is to serve as a description and executable specification of the WS protocol. This section shows how the descriptive model can be modified to be used for verification based on explicit state space exploration as supported by CPN Tools.

State space exploration [BK08b] is a technique for verification of behavioural properties of protocols with CPNs [Kri10]. This approach involves computing a directed graph where the nodes corresponds to the set of reachable states of the CPN model and the arcs represent occurrences of transitions bindings causing state changes. The construction of the state space guarantees complete coverage

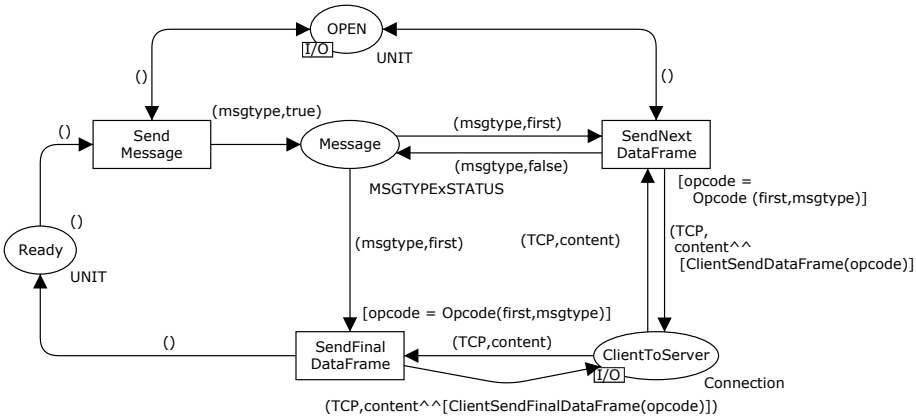


Figure 2.7: Client side sending of data frames

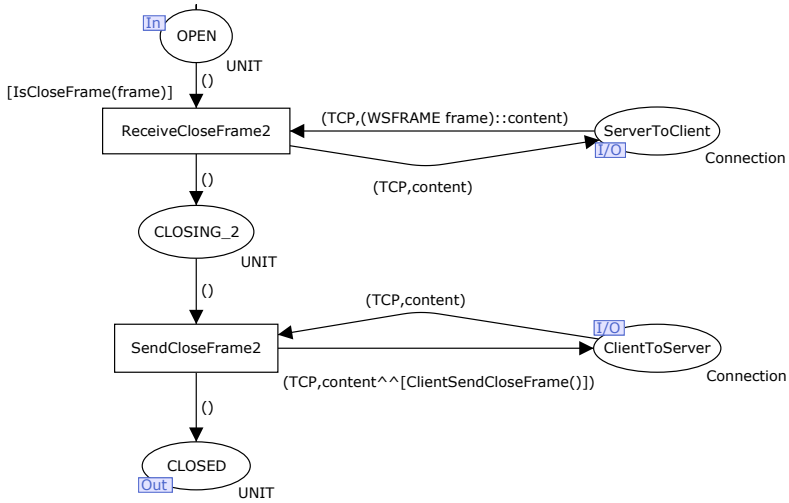


Figure 2.8: The client side of a server initiated closing handshake

of all executions and provides a highly systematic error-detection technique that make it possible to automatically (i.e., algorithmically) check whether a protocol has a formally stated desired behavioural property including properties in LTL and CTL. In addition, state space methods have the advantage that counter examples (error-traces) can be automatically synthesised if the protocol does not satisfy a given property. The main disadvantage is the inherent state explosion

problem [Val98] as the number of states quickly can become large or infinite.

The first aspect to consider in order to verify the WS model is that the WS protocol model has an infinite state space. This is because there is no bound on the number of frames that can be in transmission. In order to perform state space exploration, the descriptive model has to be modified so that there is an upper bound on the number of frames that can be simultaneously in transmission on a TCP connection.

A second element to be incorporated in the verification model is to make it possible to limit the scope of the verification. This is needed in order to make the verification process incremental, i.e., initiate the verification by considering the smallest possible configuration of the WS protocol and then gradually include more and more of the functionality. For this purpose *guards* were added to the model that made it possible to enable and disable the parts of the model concerned with e.g., sending data frames, ping frames and pong frames.

The third element was to introduce additional abstractions in the model in order to reduce the size of the state space. One example is that the descriptive model specifies that a message being sent can be of two types: binary or text. This contributes to making the state space larger, but is not necessary for analysis. Therefore, this was abstracted to a single value representing a data frame.

An initial verification of the WS protocol concentrated on termination properties of the protocol, i.e., that the connection is properly closed. The WS connection is properly closed when both the client and server are in a closed state and the TCP connection has been closed in both directions. The verification process started by considering only the open and close handshake while disabling the sending of data, ping, and pong frames. In this case, the verification model has a single terminal state representing a state where the connection is properly closed. The next configuration considered added the transmission of data. The resulting state space had a number of terminal states some of which represented states where either the server or the client went into the closing handshake *before* a message currently under transmission had been completely sent. This highlighted an issue where the WS protocol may be under-specified; a principal needs to clean up the message buffers before entering the closing handshake or prevent sending of partial messages in some other way.

Finally, the sending of ping and pong frames were included in the verification. This highlighted a second issue with the protocol specification where it is not specified how ping and pong frames should be processed during the closing handshake. This issue manifested itself by the presence of terminal states where there were frames in the TCP connection that could not be received. The model was therefore modified so that data, ping, and pong frames can be received

during the closing handshake. With this modification, it was possible to verify (for the complete verification model) that the WS connection can always be properly closed.

In addition to the two issues related to the WS specification presented above, the verification process also helped to identify a number of smaller modelling errors and thereby increasing the confidence in the correctness of the protocol model. The state spaces of all the configurations considered of the WS model had state spaces with less than 3,000 states demonstrating that explicit state space exploration can be a feasible approach also for industrial-sized protocols with the proper abstractions applied.

2.3 Related Work on CPN Protocol Modelling

CPNs have been widely used to model and verify protocols in the past. In this section we present some of the previous efforts to model and analyse protocols with CPNs. First we briefly present the results of four cases that are discussed in detail in [KS13]. Then, we discuss related work in terms of further examples of protocol modelling with CPNs .

2.3.1 The DYMO Protocol

The Dynamic On-Demand Routing Protocol for Mobile Ad-hoc Networks (DYMO) [CP07a] is a routing protocol for mobile ad-hoc networks being developed by the MANET working group of the IETF. A CPN model of the DYMO protocol was constructed in a project on modelling and validating DYMO [EKK08]. In the process of constructing the CPN model and simulating it, several issues and ambiguities in the specification were discovered. The most important issues were submitted to the IETF MANET Working Group mailing list and several of them were acknowledged by the DYMO developers and taken into account in the subsequent version DYMO specification [CP07b] (version 11).

The modelling of the DYMO protocol illustrated that the construction of a formal and executable model provides a systematic and comprehensive way of reviewing a protocol design document (such as the DYMO Internet draft) and how it can contribute to increasing the quality of a protocol design specification. Similar conclusions can also be drawn from other case studies where CPN modelling has been applied to protocols developed in the context of IETF. A CPN model of the DYMO protocol has also been developed in [BY09] where a con-

siderably more compact CPN model of the DYMO protocol directly targeting state space exploration was developed. Several additional issues related to the functionality of the DYMO protocol were reported in [BY09]. A main difference between the DYMO and WS models is in descriptiveness. The DYMO protocol is modelled as a compact encoding of a state machine and is not designed to be used to describe the protocol, but rather to verify the protocol.

Akin to the WebSocket model presented in this chapter, the DYMO model is a hierarchical CPN model. The main difference is that the DYMO CPN model is a so-called folded model. This means that a single module models all the protocol actors. This is appropriate since all actors in DYMO are peers and have the same behaviour. For client-server protocols like the WebSocket protocol this is not the case, even though there are similarities between the actors in WebSocket as well. Another difference is that the network channel is modelled with a separate module in the DYMO model. This is sensible in many situations and was adopted for our modelling approach for code generation that is presented Chap. 4.

2.3.2 Generic Access Network Architecture

The GAN protocol architecture [3GP07] is developed by the 3rd Generation Partnership Project (3GPP) for accessing telephone services via Internet Protocol (IP) networks. In the project [FK09] CPN modelling and state space exploration were used at TietoEnator Denmark in early phases of developing an implementation corresponding to a particular instantiation [Gri06] of the generic GAN architecture [3GP07] aimed at integrating IP and telephone services.

As part of the construction of the GAN model, the support for interactive simulation in CPN Tools was used to perform detailed checks to ensure that the model behaviour was as desired. Even though the use of interactive simulations (and simulation in general) cannot be used to prove correct behaviour, it proved to be very useful in identifying situations related to improper manipulations of the entries in the routing tables and security policy database - or when additional detail not present in the GAN specification had to be worked out and specified. Furthermore, interactive simulation was helpful in identifying issues that led the GAN connection establishment procedure to terminate prematurely, e.g., because a certain phase of the connection establishment was missing in the CPN model.

The interactive simulation was in later phases replaced with automatic simulation where a number of random executions of the CPN model were performed with the purpose of checking whether the execution of the CPN model resulted

in a state in which the GAN connection was properly established. Eventually state space exploration of the CPN model was conducted which succeeded in establishing the key property that a GAN connection will eventually be established provided that the GAN controller does not keep rejecting the connection request. The verification also illustrated the general observation that, in many cases, the use of basic state space exploration and the state space report (i.e., investigating standard behavioural properties of Petri nets) are sufficient in establishing key properties of a protocol design. In this case, the state space was small and could be generated in a few seconds without the use of advanced state space exploration techniques.

The GAN model is also a hierarchical CPN model. The top-level module is also set up to show the actors and channels, but also shows the network architecture which we have not adopted in our modelling approach. Like the WebSocket protocol, the GAN protocol model is not folded, meaning that the protocol agents are explicitly shown in the structure.

2.3.3 CPN Model of the RIP Protocol

The RIP protocol, developed at Ericsson Telebit A/S, enables routing of IP packets between core IP networks and mobile ad-hoc networks. The RIP case study [KWN05] used *application-specific behavioural visualisation* on top of CPN models to obtain a first executable prototype of the protocol design allowing for early experiments and for presentation to customers and management with the aim of soliciting protocol design requirements.

In the routing interoperability project, the BRITNeY Suite animation framework [WL06] was used to create an *animation GUI* on top of the CPN model. The animation GUI allows a user to observe the execution of the constructed CPN model using a graphical representation of the network architecture. The graphics is updated by the underlying CPN model according to the execution of the formally specified protocol, and the CPN model is also able to react to stimuli provided by the user via the animation GUI.

The CPN model combined with the animation GUI that was developed in the RIP project served as an early model-based executable prototype. The domain specific graphical user interface (the animation GUI) made it possible to explore and demonstrate the design of the interoperability protocol with the underlying formal model being transparent for the observer and the demonstrator. In particular, it made it possible for persons without knowledge of the CPN modelling language to experiment with the proposed design. The use of an animation GUI on top of the CPN model has the advantage that the behaviour observed by the

user is as defined by the underlying model that formally specifies the design. The alternative would have been to implement a separate visualisation application totally detached from the CPN model. This would have led to double representation of the dynamics of the interoperability protocol which could in turn lead to inconsistencies between the two representation of the design.

Another advantage offered by the development of a model-based prototype is easier to control than a physical prototype, in particular in the case of mobile nodes and wireless communication where scenarios can be very difficult to control and reproduce. The use of a model means that there is no need to invest in physical equipment and there is no need to set up the actual physical equipment early in the project. The use of a model also makes it possible to investigate larger scenarios, e.g., scenarios that may not be feasible to investigate with the available physical equipment. An additional general advantage of the approach taken in the RIP project is that at an early stage of development, the implementation details can be abstracted away and only the key part of the design have to be specified in detail. As an example, the CPN model of the interoperability protocol abstracted away the routing mechanisms in the core and ad-hoc networks, and the mechanism used for distribution of advertisements. Instead, the service assumed from these components for the interoperability protocol to work was modelled. The possibility of making abstraction means that it is possible to obtain an executable prototype without implementing all the components and thereby making validation and verification feasible.

The RIP protocol model is developed as a model-based prototype. In our WebSocket model the focus has been on describing the operation of the protocol and its principal actors. This means that we have not made more high-level views on the WebSocket protocol such as was done for RIP. Instead, the focus has been on creating the model in a descriptive manner so that the model itself can be used to convey the operation of the protocol.

2.3.4 The Edge Router Discovery Protocol

The Edge Router Discovery Protocol (ERDP) was modelled in a design project conducted at Ericsson Telebit A/S [KJ04]. ERDP is an IPv6-based protocol allowing edge routers to configure gateways in mobile ad-hoc networks with IP address prefixes. The ERDP case study used CPN modelling, state space exploration, and behavioural visualisation to identify and resolve design issues and errors during ERDP development.

The ERDP project highlighted the benefits of formal modelling and validation. Furthermore, the project emphasised the benefits of the model construction

phase which is often underestimated (or not reported) in literature on protocol validation. In the ERDP project, the modelling phase itself lead to significant insight into the protocol design, and contributed to a simpler and more complete protocol design. The construction of a CPN model and subsequent state space exploration can be seen as a very thorough and systematic way of reviewing the ERDP design specification. The project showed that the process of constructing a CPN model based on the ERDP specification provided valuable input to the ERDP design, and the use of simulation added further insight into the operation of the protocol. State space exploration, starting with the simplest possible configuration of the protocol, identified additional errors in the protocol. The results from state space exploration also demonstrate that errors are often present in the smallest configurations of a protocol system.

Overall, the application of CPNs in the development of ERDP was considered a success for three main reasons. Firstly, it was demonstrated that the CPN modelling language and supporting computer tools were powerful enough to specify and verify a real-world protocol being developed in an industrial project, and that integration into the conventional protocol development process is not difficult. Secondly, the act of constructing the CPN model, executing it, and discussing it led to the identification of several non-trivial design errors and issues that, under normal circumstances, would not have been discovered until, at best, the implementation phase. Finally, the effort of constructing the CPN model and conducting state space exploration was represented by approximately 100 person-hours. This is a relatively small investment compared to the many issues that were identified and resolved early as a consequence of constructing and analysing the CPN model.

The ERDP protocol model is somewhat similar to the WebSocket model in that it is hierarchical model where the protocol agents are explicitly modelled. As with the WebSocket model, the ERDP model also underwent analysis through state-space exploration. The ability to perform state-space exploration on code generation models is an important goal of our approach in order to support verifiability.

2.3.5 Further Examples of CPN Protocol Modelling

The Datagram Congestion Control Protocol (DCCP) developed by the IETF has been investigated in [BVA08]. DCCP is intended to provide an unreliable transport service with congestion control mechanisms. The work in [BVA08] was done in parallel with the development of the emerging DCCP standard, and concentrated on modelling and verification of the connection establishment and synchronisation procedures of DCCP. It resulted in the identification of

several functional errors in the protocol design, including discovery of deadlocks, non-progress behaviour (chatter), and problems with connection establishment in relation to sequence number wraps. The formal validation resulted in the IETF working group making small (but important) changes to the connection establishment and synchronisation procedures of DCCP. The work in [BVA08] presents two CPN models of DCCP. One is a mixture of state and event-based models. The other model is a model based on the procedures of the protocol, and is, in that respect more similar to our readable model. The paper concludes that the procedure based model, while being less compact, is more readable. Thus, the procedure based CPN model is better as a descriptive model. We have also been inspired by the procedural style while creating the PA-CPN net class for code generation in order to preserve readability and descriptiveness. The work in [BVA08] also included the development of a formal service specification for DCCP [GBVAK07] and application of the sweep-line method [VABG08] for on-the-fly checking of the protocol conformance to the developed service specification.

The classical Transmission Control Protocol (TCP) has also been modelled and verified using CPNs [BH07]. Similar to the work on DCCP, this work concentrated on the connection establishment procedures. It resulted in verifying the absence of deadlocks and live-locks in connection establishment, and a detailed specification of the circumstances under which TCP connection establishment may not be successful. Another example of transport layer protocol modelling and validation can be found in [VA08] which considers the Stream Transmission Control Protocol (SCTP).

The Internet Open Trading Protocol (IOTP) designed to provide an interoperability framework for Internet commerce was formally modelled and validated using CPNs in [OKB02b, OKB02a, OB04]. IOTP is designed to handle common trading procedures and encompass trading roles such as consumer, merchant, payment handler, and delivery handler. IOTP is organised around a collection of eight baseline transactions consisting of Purchase, Refund, Value exchange, Authentication, Withdrawal, Deposit, Inquiry, and Ping. These transactions comprise a minimal set of transactions for an Internet commerce protocol. A formal specification of the service provided by IOTP was developed using CPN in [OKB02b]. The service was specified in the form of a finite-state automaton labelled with service primitives. The automaton was extracted from the state space of the CPN model by identifying the transition bindings corresponding to service primitives of the protocol. A CPN model of the IOTP protocol itself was presented in [OKB02a, OB04]. State space exploration focused on the termination properties and absence of live-locks in the IOTP transactions. The use of state space exploration revealed deficiencies of the IOTP related to termination of transactions. A verification of the IOTP protocol CPN model [OKB02a, OB04] against the formal service specification from [OKB02b] was

presented in [OB03]. Finite-state automata language comparison was used as the criterion for conformance following the methodology of [BGH04a]. Application of an advanced state space verification technique, the sweep-line method, on IOTP was investigated in [GOBK04] exploiting an inherent progression from the start of an IOTP transaction to termination of the transaction.

The Wireless Application Protocol (WAP) has been considered in [GB00, GKB02]. WAP is designed to provide Internet services to a wide range of hand-held devices. The work of [GB00, GKB02] concentrates on the Wireless Transaction Protocol (WTP) which is an important element of the WAP architecture and protocol suite. The work in [GB00] presents a formal modelling of the WTP service and a formal modelling of the WTP protocol. Checking the conformance of the WTP protocol against the WTP service was done using finite-state automata language comparison. This approach succeeded in detecting several inconsistencies between the protocol and the service which was provided as input to the WAP forum responsible for the development of WAP. In [GKB02], the sweep-line method was used to alleviate the state explosion problem and allow for the verification of larger configurations of WTP. The application of the sweep-line method allowed configurations with parameter settings of re-transmission counters corresponding to the recommended setting for GSM and IP network to be verified. The CPN model for WTP consist of several non-hierarchical modules. This makes the WPT less descriptive since it only contains a single level of abstraction. This is in contrast to our descriptive model where we have several levels in order to allow the reader to read the model at several abstraction levels.

The Session Initiation Protocol (SIP) is a widely used protocol for the establishment of Internet multimedia session, and has been subject to formal modelling and validation in [Liu09, DL08]. The INVITE transactions have been formally analysed using state space exploration in [Liu09, DL08] leading to identification of undesired terminating states of the protocol when operating over an unreliable communication medium. Security aspects of SIP have been investigated in [Liu10]. The work of [GH06] focuses on the formal modelling of a SIP-based protocol for multi-channel service oriented architectures. A formalisation of SIP with the purpose of providing a framework model for present architectures in mobile computing is presented in [GH07]. Another multimedia control protocol, the Capability Exchange Signalling (CES) protocol, has been formally modelled using CPNs and verified using state space exploration in [LB07]. The work on the CES protocol led to the identification of protocol errors in presence of sequence number wrap. Suggested changes were incorporated in a revised CPN model, and it was formally verified showing that the discovered errors have been eliminated.

The NEO protocol which is part of the distributed transactional object database management system NEOPPOD was investigated using high-level Petri Nets in

[CDE+10]. The Coloane environment was used for the construction of the models, and verification was performed using the CPN-AMI and Helena tools that uses models that are similar to CPNs. The NEO protocol is used to coordinate data storage and retrieval in a decentralised and distributed system where data can be stored on a number of data nodes and data is accessed through the primary master node. The focus of [CDE+10] was on the protocol used for the election of the primary master node. The model of the election part of the NEO protocol consisted of eighteen modules. Since there existed no specification document for the protocol, the Petri net model was reverse-engineered from a prototype implementation. The validation process which relied on the use of state spaces discovered two flaws in the implementation of the protocol. The first flaw was discoverable through simulation. The problem arises when the network link between two nodes goes down. Then they may both elect themselves as the primary master which violates a requirement that there only exists a single primary master. The second flaw was that there exists a possibility that protocol would never terminate. This was, however, not a problem in the implementation where the problem was avoided by a side effect of programming language that is not described in more detail in [CDE+10]. Details on the flaws were provided to the software engineers responsible for the implementation of the NEO protocol.

The model used in studying the NEO protocol is another type of high-level Petri nets known as symmetric Petri nets. These models share many qualities with the CPNs but they use different languages to define colours, inscriptions and guards which is created with SML in CPN Tools but is created using a custom language in CPN-AMI models. The NEO model was constructed by reverse-engineering the prototype implementation of the NEO protocol. The purpose of the model was analysis rather than being used as a descriptive model.

The Resource Reservation Protocol (RSVP) was formally modelled and verified in [VB03b, VB03a]. The modelling and verification concentrates on verifying the absence of deadlocks and live-locks in relation to the set-up, maintenance and path release procedures of RSVP. In addition, a number of RSVP specific behavioural properties were investigated which considered in detail the internal state of the sender, router, and receiver protocol entities of the protocol. The main contribution of the work was the development of a formal specification of the RSVP path procedures. Another example on the modelling of routing protocols can be found in [Lak09] which uses Mobile Petri Nets to construct a formal model of the Mobile IP protocol. Mobile IP allows transport layer connections to be preserved when mobile nodes change their point of attachment to the Internet.

CPNs have also been used for the verification of security protocols. Privacy enhancing protocols were considered in [SOSF09], and [Gor08] addresses the

modelling and validation of PANA Authentication and Authorisation Protocol. Examples of protocols for which parametric verification has been pursued in the context of CPNs can be found in [GB06, GB05].

2.4 Summary and Contributions of Papers

The contribution of [KS13] is to provide an overview of how CPNs have been applied for modelling and validation of protocol designs. This was accomplished by presenting selected parts of CPN models and associated results from projects that had been conducted in an industrial context and with industrial sized protocols.

The paper surveys four major projects where CPNs have been used to model protocol elements and improve protocol specifications, verify models and protocol properties through state space exploration, and rapidly construct a prototype of the protocol design through behavioural visualisation combined with a CPN model. The paper also briefly presented further projects where CPNs had been used for protocol modelling and validation.

Another main reason for carrying out the survey was to gain knowledge about significant cases where CPN models had been used to model and verify protocols. This was useful in creating a code generation approach where we could build upon previous works to make it suited for generating implementations of real protocols while preserving important aspects of the models such as readability, descriptiveness and verifiability.

A main contribution of [SK12] is to describe a modelling approach based on the concept of a *descriptive specification model* which can serve as a common origin model for deriving *verification* and *code generation models*. A descriptive model, in this context, is a model that has as its main purpose to convey the operation of a protocol clearly and precisely. By abstractions and restrictions of the scope of the model, the paper shows how a descriptive model can be transformed into a model suited for verification. The descriptive model can also be transformed into a model suited for implementation via the addition of *code generation pragmatics* and by means of refinement.

A hierarchical CPN model, such as the descriptive WebSocket model, is able to show the operation of a protocol at different levels of abstraction from the protocol architecture through the major components down to the specific component behaviour. This is important both for understanding the protocol as a whole as well as allowing different stake-holders to focus on the appropriate

levels of abstraction. An important feature of a descriptive model is a high level of readability. This means that it should be easy for human readers to read and understand the model and, with the help of the model, understand the protocol. A descriptive model should also include all the important parts of the protocol as well as all the major states the protocol may be in. This is important to ensure that the model can be used as a basis for deriving implementation and verification models and also such that the descriptive specification model can be used to understand and communicate the operation of all the major parts of the protocol.

Another important contribution of [SK12] was to subject the WS protocol to formal modelling and verification which, to the best of our knowledge, had not been done before. The initial verification of the WS protocol identified some minor omissions in the protocol specification related to the closing of connections during message transfer and unspecified receptions of data, ping, and pong frames during the closing handshake. However, with proper modifications to the verification model, we were able to verify that the protocol ensures correct termination of connections.

In summary, the work presented and discussed in this chapter allowed us to have both a starting point and a goal for our code generation approach. The starting point was a descriptive WS protocol CPN model and the ultimate criteria for success of the project would be to generate code for the WebSocket protocol. Furthermore, we use the experiences gained though the survey of significant uses of CPNs for modelling and verifying protocols and by creating the descriptive WebSocket model when we define our own modelling methodology be creating a new class of CPNs in Chap. 3.

Chapter 3

Pragmatics Annotated Coloured Petri Nets and Code Generation

This chapter describes our approach to generate code for communication protocol software based on CPN models. In particular, this chapter introduces Pragmatics Annotated Coloured Petri Nets (PA-CPNs), which is a sub-class of CPNs that we have developed as a part of our approach to facilitate code generation.

This chapter is based on the following papers:

- [SKK13a] K.I.F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.

- [SKK14] K.I.F. Simonsen, L.M. Kristensen, and E. Kindler. *Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification*. DTU Compute-Technical Report-2014. Technical University of Denmark, 2014.

In this chapter PA-CPNs are introduced using the WebSocket protocol rather than the simple framing protocol that was used in the paper [SKK13a]. In [SKK14] a formal definition of PA-CPNs is presented. However, in this chapter we only introduce PA-CPNs informally. The reader is referred to the paper [SKK14] for the details on the formalisation of PA-CPNs.

3.1 Pragmatics Annotated Coloured Petri Nets

Pragmatics Annotated Coloured Petri Nets (PA-CPNs) are a class of CPNs that combine an overall model structure with *code generation pragmatics*. The class is created to allow for code generation without sacrificing verifiability or descriptiveness of the model. PA-CPNs are hierarchical models with three levels. The top level is called the *protocol systems level* and describes the overall architecture of the protocol, at a high level of abstraction. The *principal level* shows what services each principal provides as well as the life-cycle related to the invocation of the services. Finally, the *service level* specify the operation of each of the services.

The need for a new CPN class arises from the difficulty of translating general CPNs to the control-flow style of most programming languages. With PA-CPNs it is possible to create protocol models in a structured way so that they concisely describe the principals, services and operation of each service at prescribed hierarchical levels. The predefined hierarchical structure of these models also aids in making them descriptive by showing the various elements at different levels of abstraction. The models are also amenable to verification due to the inherent progress in PA-CPNs. Finally, with the help of code generation pragmatics, we are able to use these models for code generation. The CPN modules presented in this section have been selected to give an introduction of the key concepts of PA-CPN models.

3.1.1 Pragmatics

Pragmatics are syntactical annotations that are associated with CPN model elements (e.g., places and transitions). The primary purpose of the pragmatics is to add enough details to the model for generating code from them. In models we write pragmatics inside $\langle\langle\rangle\rangle$. This is similar to the notation used for UML stereotypes [Grob], which are annotations on UML elements and an inspiration for pragmatics. The name, pragmatics, reflects that they describe effects of the attached model elements that they are attached to and convey domain

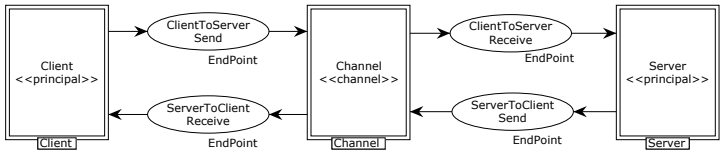


Figure 3.1: The protocol system level

specific, but not implementation specific, details about how the element should be interpreted by a code generator. The name also links to what C.A. Petri called the *pragmatic context* and *pragmatics* [Pet77] of the model elements.

The pragmatics fall into three categories: structural, control-flow, and operation pragmatics. Structural pragmatics are used to identify the structure of the PA-CPN model. These are pragmatics that specify the principal actors of the protocol, the channels, and the services provided by the principal actors. Control-flow pragmatics indicate the control-flow of principals and services by specifying when services may be invoked and the control-flow inside each service. Finally, operational pragmatics are used in service level modules to specify operations, such as sending or receiving a message, that should be carried out at certain points in the control flow.

Our approach is extensible in that it allows the modeller to add new pragmatics if required by the specific protocol or protocol sub-domain under consideration. Since pragmatics are syntactical, they have no semantic effect on the behaviour of the model.

3.1.2 Protocol System Level

Figure 3.1 shows the top-level module of the PA-CPN model of the WebSocket protocol which constitutes the *protocol system level*. The purpose of the *protocol system level* is to specify the *protocol principals* and the *channels* connecting them. Figure 3.1 has three substitution transitions named Client, Channel, and Server. Client and Server represent the two principals of the protocol. Channel represents a channel between them. We use the structural ⟨⟨principal⟩⟩ pragmatic to specify which substitution transitions represent protocol principals, and the structural ⟨⟨channel⟩⟩ pragmatic to specify substitution transitions representing channels. The places connecting the principals to the Channel are implicitly considered *channel places*. Messages (tokens) that are added to and removed from these places are considered to be sent and received.

In our modelling methodology, we require that there is exactly one protocol system module and that this module consists of one or more substitution transitions representing principals. A socket place at the protocol system level can be connected to one principal substitution transition and one channel substitution transition. This requirement is needed since we use the socket places connecting principals and channels to identify which channel or principal a message is intended for.

The protocol system level module is similar to the top level of the descriptive protocol model in Fig. 2.1. The main difference is the introduction of the explicit channel module. This has been done for purposes of verification and simulation of the CPN model. The channel module can be used to model different network service types provided by underlying communication channels. In the case of the WebSocket protocol, which is specified to run on top of the TCP protocol, the channel is modelled without packet loss or reordering, which is guaranteed by TCP. The channel is also modelled as preserving data integrity, which is somewhat stronger than what TCP can achieve, but is, to some degree achievable in practice by using a secure channel such as TLS.

3.1.3 Principal Level

The sub-modules of principal substitution transitions in the protocol system module constitute the *principal level modules*. Each principal level module specifies the *services* that are provided by the corresponding principal and the life-cycle of the principal. The life-cycle is modelled by specifying constraints on the order of service uses, and the state to be maintained across invocation of the services. The explicit modelling of the services constitute the API of a principal. Explicit modelling of services is also required in our method in order to generate code that can be integrated into different code contexts.

Here, we concentrate on the client principal as a representative example. Figure 3.2 shows the principal level module for the client. This module is the submodule of the Client substitution transition in Fig. 3.1. All substitution transitions on the principal level are annotated with either ⟨⟨service⟩⟩ or ⟨⟨internal⟩⟩ pragmatics.

The module has eight substitution transitions, six of which are annotated with a ⟨⟨service⟩⟩ pragmatic, and two with an ⟨⟨internal⟩⟩ pragmatic. Substitution transitions annotated with a ⟨⟨service⟩⟩ pragmatic model services that are meant to be called from software using the protocol. The substitution transitions annotated with an ⟨⟨internal⟩⟩ pragmatic represent services that are internal to the protocol and not meant to be called from external code. We distinguish

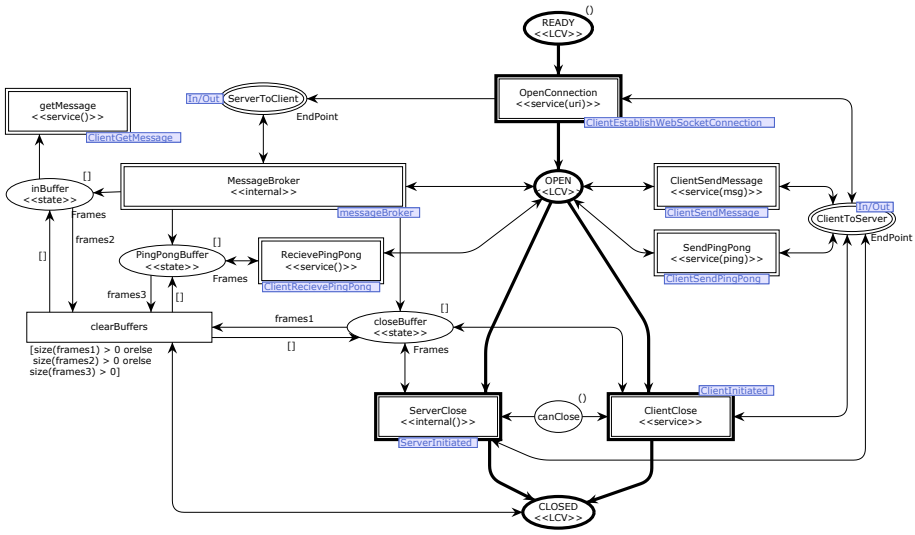


Figure 3.2: The Client module

between external and internal services in order to make it clear for users of the protocol what services are there for them and what services are there for the protocol implementation and to make the API of principals explicit in order to allow the principals to be integrated with, or called by, external code.

Some of the ⟨⟨service⟩⟩ pragmatics in Fig. 3.2 are appended with parenthesis containing parameters for the pragmatics. For services the parameters give the names of the arguments that the service expects when it is translated into code. Other pragmatics, however, may use parameters for other purposes.

Figure 3.2 has three ⟨⟨LCV⟩⟩ places, Ready, Open and Closed. The ⟨⟨LCV⟩⟩ annotated places are used to determine when in the life-cycle of the protocol the services may be invoked. In the initial state, where the Ready place is the only ⟨⟨LCV⟩⟩ annotated place containing a token, only the OpenConnection service can be invoked. After the OpenConnection service has successfully finished, the protocol enters the OPEN state by placing a token at the OPEN place. The principal stays in the OPEN state until the WebSocket connection is closed and enters the CLOSED state. While in the OPEN state, all the other services can be invoked. In this state, the Client can send and receive messages. The getMessage service is always enabled and its function depends on the message buffer, which is modelled by the inBuffer place and controls whether the service returns a message or not. The getMessage service is not bound to the OPEN state so that it can be called to get messages that have been received but may not have been

processed before the connection is closed. The places annotated by a ⟨⟨state⟩⟩ pragmatic hold data that is relevant to the operation of the principal across services. In this example, state places are used as to hold incoming messages. The messages are received via the MessageBroker internal service which will be discussed in the next subsection.

3.1.4 Service Level

The sub-modules of the substitution transitions annotated with ⟨⟨service⟩⟩ on the principal level specify the detailed behaviour of the services provided by the principal. The detailed behaviour is modelled in a control-flow oriented manner using ⟨⟨Id⟩⟩ pragmatics on places to make the control flow explicit. Modelling the services in a control flow oriented manner serves two main purposes. The first purpose is to provide for comprehensible models in that the explicit control flow aides in reading and understanding the model of the service. This is in contrast to a pure event-oriented approach to modelling [BGH04b]) from which no control flow is explicit. The second purpose of modelling in a control flow oriented manner is to automatically generate code with a structure that resembles what a human programmer would implement. This makes it easier to inspect and maintain automatically generated code, and provides code with better performance since it reflects the intended use of the constructs provided by a conventional target programming language.

As a representative example of a service level module, we consider the MessageBroker service which is shown in Fig. 3.3. At this level, the ⟨⟨service⟩⟩ pragmatic annotates ordinary (non-substitution) transitions to indicate the single entry point for the corresponding service primitive. Hence, it is possible to have only one transition annotated with ⟨⟨service⟩⟩. The message to be sent is represented by the parameter `msg` of the ⟨⟨service⟩⟩ pragmatic. Transitions representing the termination of the service are annotated with the ⟨⟨return⟩⟩ pragmatic. We require that there is exactly one transition in a service level module that is annotated with ⟨⟨return⟩⟩ and exactly one transition that is annotated with either ⟨⟨service⟩⟩ or ⟨⟨internal⟩⟩.

Places modelling the control-flow in the MessageBroker module are annotated with an ⟨⟨Id⟩⟩ pragmatic. From a control flow perspective, the MessageBroker has an overall sequence (starting at transition ReceiveDataFrame and ending at the place terminated). Inside this square there is a repeat-until loop (starting at place waitreceive and ending in place end). Inside the loop there is a branch starting at the place dispatch and merging again at place continue. The branch has four cases which distributes the messages to the right buffers based on the message type. In addition, model elements in the service level module contain

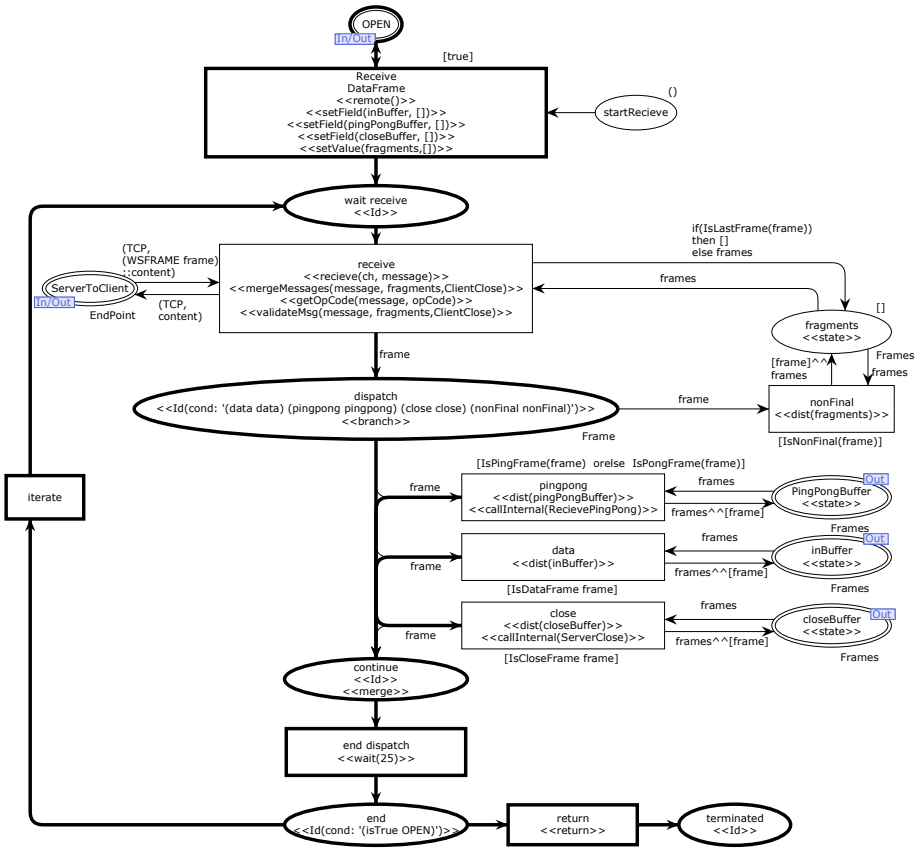


Figure 3.3: The MessageBroker Service

a number of operational pragmatics that are not discussed in detail here.

3.2 Code Generation Approach

This section outlines the code generation approach that is a central contribution of this thesis as well as the concepts behind the approach. The code generation approach is implemented in the PetriCode tool (see chapter 4) and has been evaluated as presented in Chap. 5.

The generation of code from a PA-CPN model proceeds in three phases depicted in Fig. 3.4. The first phase is to add pragmatics that can be derived from the

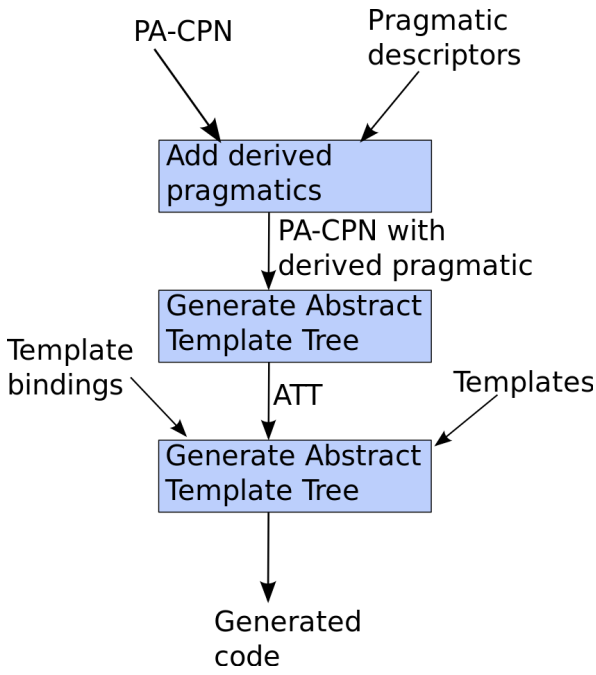


Figure 3.4: The phases of our code generation approach

PA-CPN structure to the model. This is done to provide additional pragmatics for subsequent code generation phases and to reduce the amount of annotations that modellers must add manually. This phase requires a PA-CPN model and descriptions of the available pragmatics and produces a PA-CPN with derived pragmatics added. The second phase is the construction of an Abstract Template Tree (ATT) which serves as an intermediate representation in the code generation process. The second phase binds code generation templates to the nodes of the ATT corresponding to the target platform under consideration. This phase requires a PA-CPN model with all relevant pragmatics added and produces an ATT. The third phase is to traverse the ATT and invoke the code generation templates in order to emit code. This phase requires an ATT as well as template bindings and all the relevant templates where all platform specific information is kept. Below, we illustrate the three code generation phases using the annotated messageBroker service module shown in Fig. 3.3 as an example.

In the pragmatics derivation phase we automatically compute a set of derived pragmatics that identify common control flow structures and operations, such as sending and receiving packets, or manipulating states represented in the PA-CPN model. Pragmatics are derived based on structural patterns that match

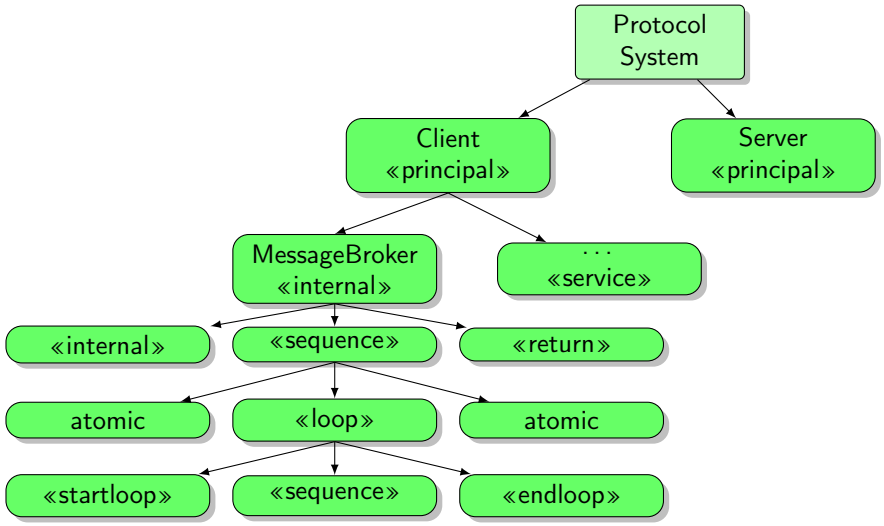


Figure 3.5: ATT of the Client and Message Broker

elements based on pragmatics and outgoing and incoming arcs and the attached elements. As an example, in Fig. 3.3, the place wait receive has been automatically annotated with a ⟨⟨startloop⟩⟩ pragmatic based on the number of incoming and outgoing arcs and the concept of progress on the control-flow path. The ability to derive pragmatics allows the PA-CPN models to be less verbose and more descriptive by not requiring the modeller to restate information that is apparent from the model structure. Furthermore, it would be possible to make derived pragmatics not visible in the model to further preserve readability.

An ATT is an ordered tree of nodes and resembles abstract syntax trees. An excerpt of the ATT of the WebSocket protocol is shown in Fig. 3.5. The two major types of nodes in the ATT are *leaf* (operation) nodes and *container* nodes. A leaf node does not have children and contains pragmatics for one or more sequential operations such as sending on a channel or accessing a state variable. A container node has in addition to associated pragmatics, an ordered list of child nodes. The root node of the ATT represents the entire protocol system. In Fig. 3.5, the root node has two children Client and Server, which represent the principal agents of the protocol. In Fig. 3.5, the Client node has the MessageBroker node which contains an ⟨⟨internal⟩⟩ pragmatic and also several other ⟨⟨service⟩⟩ and ⟨⟨internal⟩⟩ annotated nodes that are not shown in the figure. Each service module contains exactly one transition with the ⟨⟨service⟩⟩ pragmatic, which is the starting point for the method modelled by the sub-module. The subsequent set of nodes is constructed according to the control-flow of the service. These nodes are added as sub-nodes to the corresponding

service node. The next level show the entry and exit nodes that are annotated with the ⟨⟨internal⟩⟩ and ⟨⟨return⟩⟩ pragmatics, as well as a ⟨⟨sequence⟩⟩ node which contains the body of the internal service. The body of the MessageBroker contains an atomic block (leaf node), a loop block, and another atomic block. The loop, which corresponds to the loop that starts at the wait receive place and ends at the end place in Fig. 3.3, contains a startloop and endloop node surrounding a sequence.

The ATT is generated by a structural traversal of the CPN model. This traversal starts at the protocol system module and, for each ⟨⟨principal⟩⟩ pragmatic, it generates a corresponding node in the ATT. On the next level, the generator looks for modules annotated with a ⟨⟨service⟩⟩ or ⟨⟨internal⟩⟩ pragmatic and adds corresponding nodes. At the service level, the PA-CPN model is traversed from the single ⟨⟨service⟩⟩ or ⟨⟨internal⟩⟩ following the control-flow of the module and generating nodes based on a block decomposition of the module. The block decomposition of a service module creates blocks such as atomic blocks, that contain a single transition, loops, and choice blocks. Atomic blocks are leaf-nodes in the ATT and loops and choice block nodes contain the blocks that are decomposed from the sub-nets inside the loops and choice structures (see [SKK13a] and [SKK14] for further details on the block decomposition).

When the ATT has been generated, and in order to generate code for a particular platform, the pragmatics represented by the nodes of the ATT are bound to code generation templates. This is done by means of *template bindings*. Template bindings serves as a configuration by selecting the appropriate templates for each pragmatic. Generating the protocol software consists of traversing the ATT and invoking the associated templates for each node as described by the template bindings. When a pragmatic is transformed to code, its template is run through a template engine together with parameters given by the pragmatic definition and the PA-CPN structure. Finally, the code generated for each of the nodes in the ATT are combined to the full code of each of the principals. The technical aspects of how template bindings, described by *template descriptors*, and *pragmatics descriptors* have been realised are discussed in detail in Chap. 4

3.3 Code Generation from Petri Net Formalisms

In this section we discuss briefly other uses of Petri Nets for code generation. Several of the earlier code generation approaches from Petri Net models are surveyed in [EKK03] which enumerate sixteen approaches. All the surveyed approaches are simulation based which means that they, in some sense, execute the Petri Nets directly with some additional functionality added. The approaches

are also platform dependent only working with one or two programming languages.

In [Phi06], possible methods for code generation from high-level Petri Nets (HLPNs), such as CPNs, are discussed. Furthermore, the paper presents a new hybrid approach for code generation. The general methods for code generation from HLPNs are, according to [Phi06], structural analysis, simulation-based and reachability graph based. Code generation based on structural analysis is based on identifying regular patterns in a Petri Net that can be translated into programming language concepts. Simulation based approaches are based on running Petri Nets directly rather than generating some representations of the models in code. Reachability graph based methods computes all possible states of a Petri Net and executes an automaton based on the Petri Net states. The method proposed in the paper [Phi06] is a hybrid of simulation based and structural analysis methods. Here structural analysis is used to generate Java classes with empty methods based on class diagrams and a simulation based approach is used to fill the methods. In our approach, we do structural analysis (cf. derived pragmatics) but not simulation or reachability graph generation. Simulation based approaches often have performance issues while approached that are based on computing reachability graphs may require large amounts of memory to store reachability graphs.

Renew [K+04] is a tool that allows creation and execution of object-oriented Petri Nets. The Renew tool supports several modelling formalisms based on various forms of Petri Nets. Renew supports Reference nets which can be annotated with Java code and can be executed using a built-in simulator engine. The simulator can execute the nets incorporating the Java annotations in a headless mode so that no visualisation will occur. This means that the simulations can be used as stand-alone programs. The simulation approach is in contrast to our code generation approach where code is generated and can be inspected and compiled as computer programs created with traditional programming languages.

The approach in [LT07] is similar to our approach in several respects. The iterative modelling approach mandated by Coloured Control Flow Nets (CCFNs) in [LT07] is reminiscent of the control flow paths at the service level of PA-CPNs. Also, the component types in AJWNs, which is used as an intermediary representation, are similar to the blocks on our service level. Our approach differs from the approach in [LT07] in that the target domain for [LT07] is reactive systems while our target domain is protocols. Also, the approach of [LT07] is targeted exclusively to the Java platform while our template-based approach is flexible when it comes to target platforms. Also, our approach uses the same CPN class through all stages of the modelling while CCFNs need to be transformed to AJWNs before executable code is generated. Furthermore,

our pragmatics differ from the annotations of AJWNs in that our annotations are platform independent. The authors of [LT07] and we have similar requirements to the generated code. However, although we have more or less equivalent definitions of what constitutes readable code, the end results look quite differently. For example, where our approach will in-line the generated code for each pragmatic on a transition the approach of [LT07] will generate a method for each transition. This may be due to the different domains or the fact that it is difficult to pinpoint what makes code readable.

In [Mor00] the author describes a code generation approach of generating code from CPNs for an access control system. The generation takes advantage of the fact that the CPNs uses the SML programming language for all inscriptions. This means that it is fairly simple to extract the model in the form of SML code and combine it with the simulator from CPN Tools. And by using external libraries, the CPN can interact with other devices through a specialised protocol for access control systems. The paper also presents a case study of where the techniques discussed are used to generate an access control system for an industrial actor.

The approach in [Mor00] has the advantage that it can generate code from arbitrary CPNs. However, it does limit the code to where SML compilers are available. Compared to our approach this gives the modeller more freedom at the cost of being able to only generate code for a single programming language and some platform dependence. Furthermore, the approach of [Mor00] may be difficult to integrate with external code since it does not have a clearly defined API as we have in our approach. A somewhat similar approach is also taken in [KMZ+08] where the core of a tool for scheduling courses of actions is created based on a CPN model. The model is extracted from the modelling tool and executed as an SML program.

Process-Partitioned CPNs (PP-CPNs) [KW10] have been used to automatically generate code for several purposes including protocol software. PP-CPNs are a restricted sub-class of CPNs. Code is generated from PP-PCNs by first translating the PP-CPN into a control flow graph (CFG). The CFG is translated into another intermediary representation which is dependent on the target platform, and from this representation code is generated. In [KW10], PP-CPNs are used to model and obtain an implementation for the DYMO routing protocol using the Erlang programming language and platform.

Both PP-CPNs and our modelling language are sub-classes of CPNs. However, where we rely on pragmatics to control code generations, PP-CPNs rely on restricted colour sets and CPN structure to allow the generator to deduce the needed information. Our approach also models the environment of the services while PP-CPNs are geared more to just modelling the services themselves. This

allows us to represent the protocol at higher levels of abstraction on the protocol and principal levels as well as on the service level. It also allows us to define how the services should be called in a structured way by using places annotated with $\langle\langle \mathsf{LCV} \rangle\rangle$ pragmatics in principal level modules.

In [vdAJL05], an approach for generating code from CPNs for the banking sector is presented. A CPN model is translated manually into a Coloured Workflow Net (CWF). The CWF is further translated to the actual target language the Business Process Execution Language (BPEL), using a semi-automatic approach based on patterns and structural analysis.

The code generation approach we have outlined in this chapter, in contrast to the surveyed approaches, explicitly models the API of the principals at the principal level of the CPN model[1]. This is important for the integrability of the generated software. By employing a template-based approach to code generation, we also provide a larger degree of platform independence which is further contributed to by the control-flow structure of the service level. Finally, the PA-CPN structure allows models to be simultaneously descriptive, verifiable and amenable to code generation. This is not the case for all the approaches discussed in this section.

3.4 Related Protocol Modelling Languages

Many other formalisms for modelling protocol software exist. In this section we discuss some of these formalisms and relate them to our modelling and code generation approach.

There are several tools for modelling and generating protocol software based on the Specification and Description Language (SDL) [IT99, BD02]. SDL is created for the purpose of modelling protocols, and is extensively used in the telecommunications industry. The IBM Rational SDL Suite (previously Tau SDL Suite and SDT) is among the most well known proprietary tools for SDL. The Rational SDL Suite supports code generation for SDL models to C and C++ code and also supports verification through model checking. Another SDL tool is Jade [PdSD+00] that supports editing and analysis/verification of SDL models. Code generation for JADE is still in development. The SDL Integrated Tool Environment (SITE) supports editing of SDL models and code generation to Java and C++ code. SITE also supports some analysis of SDL models. SDL is a graphical language based on Finite State Machines (FSMs). This allows verification of protocols using model checking techniques. A major difference between CPNs, and by extension PA-CPNs, is that SDL uses different

[1] In [Phi06] the API is modelled using UML class diagrams.

images symbolising each operation. This makes SDL a more complex modelling language where the modeller and the readers must be familiar with the comparatively large graphical syntax of SDL rather than the few model elements of CPNs. Furthermore, the code generation offered by the Rational SDL Suite and SITE are both limited in terms of the supported platforms as opposed to our platform independent approach.

The most popular modelling language for computer systems is UML [Grob]. There has also been several efforts to employ UML to model and verify protocols. In [ALPT05], a method for creating protocol software and hardware using techniques influenced by the Model Driven Architecture (MDA) [OMG] is described. The method involves using UML [Grob] by extracting application requirements in the form of a UML Use Case Diagrams. Use Case Diagrams are then used via a number of refinements and transformations to generate a Data Flow Diagram (DFD) and a Class diagram. The DFD and the class diagrams are combined with a platform model in order to generate specific implementations. The platform model describes how different operations are implemented on a specific platform and is realised using a UML class diagram. Details are, however, not provided on how actual program code is generated using this platform model. The approach presented allows for some limited verification based on model queries, such as verifying naming conventions and supported OO-features. The approach described in [ALPT05] uses several modelling notations to model the system at different levels. This is in contrast to our approach where only one modelling notation is used to describe both the structure and behaviour of protocols. Although the approach allows for verification of some details, it does not support the same level of verification of behavioural properties that is supported by CPN models and which can easily be derived from the descriptive models that is the basis for our approach.

UML and a custom language is used in [PvKHT00] to model and generate code for protocol software. A UML profile called the Graphical Protocol Description Language is created. The UML profile relies on UML stereotypes and various UML diagram types to model static and behavioural aspects of network protocols. The models are organised into four main diagrams: the Protocol System Structure Diagram, the Protocol Interface Diagram and the Protocol Entity Diagram, and the Behaviour Description Diagram. The Protocol System Structure Diagrams, which are reminiscent of protocol system level modules of PA-CPN models, contain system elements and associations between them. The Protocol Interface Diagrams contain the messages that can be sent and received by the various protocol elements. The Protocol Entity Diagrams, which are reminiscent of the principle level modules of PA-CPNs, shows the internal structure of a protocol entity. The Protocol Behaviour Diagrams, which are similar to the service level modules of PA-CPNs, describes the behaviour of each protocol entity. In [PvKHT00] these diagrams together with a custom textual language

named GAEL are used to obtain an implementation in SDL, however, the authors conjecture, it can be used to generate implementations for any platform. Unlike our approach, the approach described in [PvKHT00] uses several diagram types are used whereas in our approach only is PA-CPNs to model protocols. This may make our approach somewhat simpler. Although the stereotypes used in [PvKHT00] can be compared to our stereotypes our approach adds flexibility by allowing the developer to add custom stereotypes. This allows the developer to incorporate concepts useful for the protocol at hand instead of relying on textual language to describe these concepts in the model itself.

Kukkala et. al. presents an article [KHHH04] describing a way to use UML 2.0 to model and implement the TUTMAC protocol, a media access protocol for wireless networks. The implementation is realised by creating a UML 2.0 model consisting of class diagrams, architectural diagrams, and state charts. Using the Tau G2 tool, the model is transformed into C/C++ code and a prototype is created and tested using a simple configuration. This approach also uses several diagram types for modelling protocols in contrast to PA-CPNs where all the model layers are CPN modules. Furthermore, the approach only support code generation for C/C++ code and may not be as platform independent as out approach.

The UML approaches tend to require several diagram types for modelling the same system. This is in contrast to our approach where we use a single modelling notation for all layers of abstraction considered. This simplifies modelling and presentation by reducing the number of diagrams types that must be created and, more importantly, studied to understand the model. These various UML diagram types are amenable to verification in varying degree. State diagrams are perhaps the most commonly used UML notation for the verification of behavioural properties since they are based on Harel state-charts [Har87]. There have also been made efforts to translate UML models into other formal languages to perform verification [BBSCdlF05, LP99, PvKHT00].

An overview of code generation for state machines is given in [DPRZ12]. The paper presents a systematic literature review of methods of automatically obtaining implementations from state machine models. The methods are divided into two types of code generation techniques: pattern based techniques and non-pattern based techniques. The pattern based techniques use various software design patterns [GHJV95] as a basis for their code generation. The techniques that are not based on design patterns uses techniques such as nested switch statements, linked lists of transitions or encoding states in Java Enums.

State machines have some similarities with CPNs, however, where the state of a state machine is given by the current state, the state of a CPN is given by the token colours present at the places in the model. Also, the modelling

languages differ in how they handle concurrency where state charts use explicit concurrent sections while CPNs are implicitly concurrent until they are explicitly synchronised. This makes CPNs arguably more convenient when modelling concurrent systems like protocols where several actors act concurrently, although there are, of course, mechanisms to support concurrency in various state machine specification formalisms as well. While PA-CPNs do put some restrictions on concurrency compared to CPNs, concurrent operation between principal agents is still supported.

Several formalisms have been used for the verification of protocols. The Promela language [Hol91] and the Spin model checker [Hol97] have been widely used to model and verify protocols. Promela, being a textual language, does not offer the same degree of readability at multiple levels of abstraction as PA-CPNs do in our modelling approach. Also, being a textual language makes Promela less appropriate as a basis for code generation using annotations since annotations in a textual language will hamper the readability of the model.

The Language of Temporal Ordering Specification (LOTOS) [ISO89, 15401, BB87] was developed as part of International Standardisation Organisation (ISO) efforts and linked to the development of the Open Systems Interconnection (OSI) reference model. LOTOS is founded on the Calculus of Communicating (CCS) [Mil89] and adds a data type component to CCS based on algebraic specification. LOTOS, just as Promela, is a textual language, which makes it a less than ideal candidate for code generation based on syntactical annotations.

The Extended State Transition Language (Estelle) [ISO] also originated from OSI standardisation efforts and is based on extended finite state machines [Boc78] combined with extensions to the PASCAL programming language. Work has also been done to define a formal semantics for Estelle by translating to Petri Nets [Cou87]. Estelle is a graphical language with several similarities to CPNs and could probably be extended with pragmatics. However, Estelle uses more types of model elements than CPNs which only used Petri Nets together with SML. This makes CPNs a more lightweight modelling language.

3.5 Summary and Contributions of Papers

The contribution of the paper *Code Generation From Pragmatics Annotated Coloured Petri Nets* [SKK13a] was to describe the code generation approach that has been developed in this thesis and to informally define pragmatics and PA-CPNs. The paper [SKK13a] describes PA-CPNs and how PA-CPN models can be used for code generation by first automatically deriving additional

pragmatics based on the PA-CPN structure. Then ATTs and the process of generating them is described. And finally, the paper [SKK13a] describes how pragmatics stored at ATT nodes can be used for code generation and how the code generated for each ATT node can be combined to form the code for an executable program implementing the protocol under consideration. The paper used a simple framing protocol as an example. In this chapter, for the sake of consistency within the overview part of this thesis, we used the WebSocket protocol as an example.

PA-CPNs are inspired by the descriptive WebSocket model described in Chap. 2 and the other CPN models surveyed in Chap. 2. We have chosen to use a hierarchical structure of PA-CPN models, however, in contrast to the descriptive WebSocket model, PA-CPNs require a set number of hierarchical levels. The service level in PA-CPNs is inspired by the procedural style found in [BVA08].

The contribution of *A Formal Definition of Pragmatic Annotated Coloured Petri Nets for Automated Protocol Software Generation and Verification* [SKK14] is to provide a formal definition of PA-CPNs and parts of the code generation approach. The paper also discusses the use of advanced verification techniques on PA-CPNs where we show that we can exploit properties of PA-CPNs to make verification more efficient. The verification aspect of this paper is discussed in Chap. 5.

Chapter 4

PetriCode: Tool Support for Code Generation

This chapter introduces the PetriCode tool. PetriCode is an implementation of the code generation approach that was presented in Chap. 3 and has been used to perform the evaluation of our code generation approach as presented in Chap. 5.

This chapter is based on the following paper:

- [Sim14b] K.I.F. Simonsen. PetriCode: a tool for template-based code generation from CPN models. In *Software Engineering and Formal Methods*, pages 151–163. Springer, 2014.

The discussion in this chapter is mainly concerned with the architecture and technical design choices of PetriCode since the code generation approach has already been discussed in Chap. 3.

4.1 Architecture and Design of PetriCode

PetriCode is an implementation of the code generation approach described in Chap. 3 which is based on the following key concepts:

- **Pragmatics** which are annotations used to guide the code generation and specify operations.
- **PA-CPNs** which is the CPN class we have defined to serve as the basis for code generation.
- **Pragmatics Descriptors** which describe the available pragmatics.
- **ATTs** which are intermediary representations created based on PA-CPNs and used to generate the final code.
- **Template Bindings** which are used to bind pragmatics to code generation templates. We are able to generate code for different platforms by replacing the template bindings to bind pragmatics to templates written for a different platform.
- **Code Generation Templates** which are applied to pragmatics based on the template bindings.

A number of design choices was made in order to make PetriCode fulfil the requirements for our code generation approach and being a flexible tool for its users and maintainable for developers. When designing and implementing PetriCode, there were a number of key requirements that needed to be addressed and which affected the choice of software technologies used for the implementation.

An important requirement is the ability to read, parse and write CPN models stored in the format used by CPN Tools [JKW07]. The Access/CPN [WK09] library provides this capability for the Java platform. Therefore, in order to use Access/CPN it is necessary to choose a platform with good integration with Java libraries. Furthermore, in order to accommodate pragmatics it is required to be able to add pragmatics to Access/CPN meta models. Another important requirement was to easily be able to create Domain Specific Languages (DSLs) for defining pragmatics descriptors and template bindings. Finally, in order to make sufficient progress given the assigned time-frame for this project, development speed was also a priority in our design choices.

Given that we wished to utilise Access/CPN, the available programming languages were limited to languages that run on the JVM. The Groovy program-

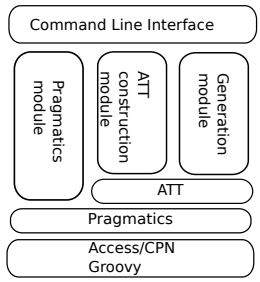


Figure 4.1: PetriCode Architecture

ming language [Groa] was chosen as the main programming language and platform for PetriCode. Groovy, which runs on the Java Virtual Machine (JVM), was chosen because it has seamless integration with all Java libraries including Access/CPN. The Groovy programming language also has a simple mechanism (not available in Java) to manipulate classes at runtime, making it a simple matter to add pragmatics support to Access/CPN, and also supports many types of DSLs. Groovy also has additional useful features such as a command-line interface options builder and a powerful template engine that can be used for code generation purposes. Another important reason for choosing Groovy was that we had experience using Groovy in previous projects [SMR10].

Two other popular languages that run on the JVM are Java [GJSB05] and Scala [O+08]. Neither Java nor Scala were chosen even though it would be possible to use either of these languages to develop PetriCode. However, neither of these languages provide facilities for meta-programming so it would be more cumbersome to extend Access/CPN with pragmatics than it was using Groovy. In addition, Java, does not have features that help to create DSLs in a simple manner. Scala, however, does have good support for DSLs. Meta-programming for the JVM comes at some cost in terms of the speed of programs. In this project we judged that ease of development was more important than the speed of code generation. Another alternative, JRuby [NES+11] which is an implementation of Ruby for the JVM, does have meta-programming facilities and good supports for DSLs. However, interaction with Java libraries such as Access/CPN is not as simple as in Groovy because of differences between the class and object structure of Ruby [MI02] and Java.

Figure 4.1 provides an architectural overview of PetriCode. PetriCode is controlled by its main class PetriCode which makes up the Command Line Interface module of the application and controls the code generation process. PetriCode parses the command-line arguments and calls the modules shown directly below the Command Line Interface in Fig. 4.1 as appropriate. All the modules de-

pend on Access/CPN for reading and manipulating CPN models. As explained above, PetriCode is implemented using the Groovy language and builds upon the Groovy and Java platforms. All modules are dependent on the data model for pragmatics. The ATT and Generation modules also share a data model for ATTs.

The overall program flow of PetriCode is shown in Fig. 4.2. Each column of the flow chart represents a module of PetriCode. The left column is the pragmatics module, the middle column is the ATT module, and the right column is the generation module. Code generation begins with reading and parsing a CPN model and the pragmatics. The next step in the code generation process is to derive pragmatics that do not have to be added manually, but instead they can be automatically derived from the structure of the model. After all the derived pragmatics have been added to the PA-CPN model, the generation process, optionally, checks the pragmatics with regards to given constraints. This is optional because checking that all pragmatics are defined in pragmatics descriptors may slow down development in early stages of a project. After checking the pragmatics, the code generation process enters the ATT module where the first step is to generate the ATT. Then, the ATT can optionally be output in either an XML format or as a picture. Based on the command-line arguments given, the process optionally terminates after generating and outputting the ATT. If the generation process is to continue, the code generator will generate code appropriate for each node of the ATT. Then, the generated code fragments are combined in a bottom-up fashion until the code for each principal has been completely generated. Finally, the code is written to files and the process is terminated.

4.2 Pragmatics Descriptors and Template Bindings

In order to specify the available pragmatics and bind them to templates PetriCode uses DSLs. The use of DSLs was a design choice that allows a high degree of flexibility for protocol designers to easily extend PetriCode with new pragmatics and templates. Furthermore, the DSLs make it simple to extend PetriCode by adding new functionality to the DSLs. By DSL we mean, in this thesis, any programming language that is designed for a specific problem domain regardless of how the DSL is implemented.

The pragmatics description language is a DSL that describes the available pragmatics. A core set of pragmatics (see the examples in Fig. 4.3 and Fig. 4.4) is provided by PetriCode while others may be provided by the user using the prag-

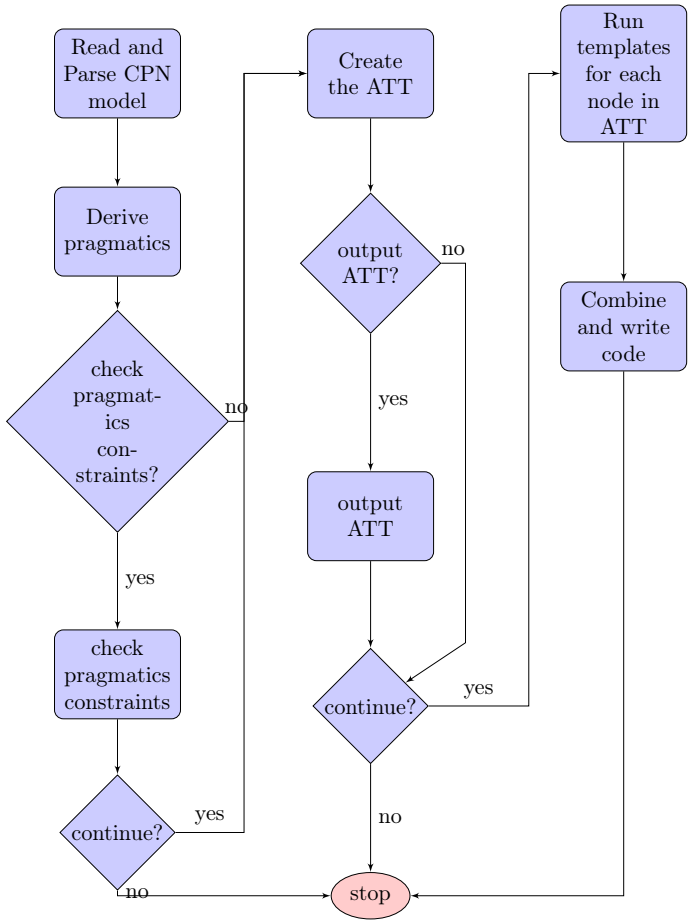


Figure 4.2: Control flow of PetriCode

matics description language. The language consists of *descriptors* that describe a pragmatic. Each descriptor consists of the name of the pragmatic followed by a pair of parenthesis. Inside the parenthesis, the parameters of the pragmatics definition are given in the form of key-value pairs. The parameters for a pragmatics descriptor are `origin`, `constraints` and `derivationRules`. The `origin` parameter indicates whether the pragmatic is explicitly given by the modeller or should be automatically derived. The first pragmatic descriptor in Fig. 4.3 describes the $\langle\langle$principal$\rangle\rangle$ pragmatic. The origin field of $\langle\langle$principal$\rangle\rangle$ indicates that this is an explicit pragmatic meaning that it will not be derived automatically. The `derivationRules` parameter gives the patterns that should be used to find the elements of a CPN model where a derived pragmatic should be

```
principal(origin: 'explicit', constraints: [levels: 'protocol',
          connectedTypes: 'SubstitutionTransition'])

channel(origin: 'explicit')

id(origin: 'explicit', controlFlow: true, constraints:
           [levels: 'service', connectedTypes: 'Place'])
lcv(origin: 'explicit')

service(origin: 'explicit', constraints:
        [[levels: 'principal', connectedTypes:
        'SubstitutionTransition'],[levels: 'service',
        connectedTypes:'Transition']])

state(origin: 'explicit', constraints: [levels:
      ['principal','service'], connectedTypes:'Place'])

return(origin: 'explicit', constraints:
        [levels: 'service',connectedTypes:'Transition'] )
```

Figure 4.3: The explicit core pragmatics for PetriCode

added. The last pragmatic descriptor in Fig. 4.4 is the ⟨⟨endLoop⟩⟩ pragmatic. The ⟨⟨endLoop⟩⟩ pragmatic has derivation rules that states that it should be added to nodes already annotated with ⟨⟨Id⟩⟩ , with at least two outgoing edges and exactly one back-link (a connection to somewhere earlier in the control flow path). In addition, both the ⟨⟨principal⟩⟩ and ⟨⟨endLoop⟩⟩ pragmatics have some constraints indicated by the `constraints` field. This means that the pragmatics should only reside on places where the constraint is fulfilled. In the case of the ⟨⟨principal⟩⟩ pragmatic this means that it should only be used to annotate nodes on the protocol system level which are substitution transitions.

Template bindings are also specified in a domain specific language (DSL). An extract of the template descriptor for generating Groovy code covering three of the pragmatics from Fig. 3.5 can be seen in Fig. 4.5. Each line of the template descriptor consists of a name followed by a left-parenthesis followed by key value pairs where the keys are `pragmatic` which contains the name of the pragmatic, and `template`, which contains the path to the template, `isContainer` which indicates whether this pragmatic denotes a container, or `isMultiContainer`. The multi-container flag is primarily an implementation detail that is used internally in Petricode to indicate whether the container is of type loop or choice.

Generating the protocol software consists of traversing the ATT and invoking the associated templates for each node as described by the template binding.

```
branch(origin: 'derived', derviationRules:
    ['new PNPattern(pragmatics: [\'Id\'], minOutEdges: 2,
    backLinks: 0, forwardLinks: 0)'],block:
    [type: "branch", ends: "merge"],
    constraints: [levels: 'service', connectedTypes: 'Place'])

merge(origin: 'derived', derviationRules:
    ['new PNPattern(pragmatics: [\'Id\'], minInEdges: 2,
    backLinks: 0, forwardLinks: 0)'],
    constraints: [levels: 'service', connectedTypes: 'Place'])

startLoop(origin: 'derived', derviationRules:
    ['new PNPattern(pragmatics: [\'Id\'],
    minInEdges: 2, forwardLinks: 1)'],
    block: [type: "Loop", ends: "endLoop"],
    constraints: [levels: 'service', connectedTypes:'Place'])

endLoop(origin: 'derived', derviationRules:
    ['new PNPattern(pragmatics: [\'Id\'],
    minOutEdges: 2, backLinks: 1)'],
    constraints: [levels: 'service', connectedTypes:'Place'])
```

Figure 4.4: The derived core pragmatics for PetriCode

```
internal(pragmatic: 'internal',
         template: '.../plattforms/groovy/externalMethod.tmpl')
startLoop(pragmatic: 'startLoop', template: 'groovy/loop.tmpl',
          isContainer: true, isMultiContainer: true)
```

Figure 4.5: Extract of binding descriptor for the Groovy platform

```
%%VARS:__LOOP_VAR__%%
__LOOP_VAR__ = true
while(__LOOP_VAR__){
    %%yield%% }
```

```
${params[0]} << message
%%VARS: ${params[0]}%%
```

Figure 4.6: Examples of templates for loops (left) and dist (right)

When the pragmatics attached to a node in the ATT are transformed to code, the corresponding templates are run through the template engine together with a number of parameters given by the pragmatic definitions and the PA-CPN structure. The templates are combined by replacing a special tag in the container templates, `%%yield%%`, with the text of the generated for the children of the ATT node with the container template.

As an example of a template, the template for the loop pragmatic for the Groovy language is given in Fig. 4.6 (left). The template creates a while-loop which continues while the `__LOOP_VAR__` variable is true. The body of the loop is populated by replacing the `%%yield%%` directive with the code generated by the templates of the sub-nodes in the ATT. `__LOOP_VAR__` is updated at the end of the loop by the ⟨⟨endLoop⟩⟩ pragmatic. The `%%VARS:__LOOP_VAR%%` is used to tell the code generator that the `__LOOP_VAR__` is used in this template and should, depending on the platform and programming language, be declared. The ⟨⟨dist⟩⟩ pragmatic which is present on several transitions in Fig. 3.3 is an example of an operation pragmatic. The ⟨⟨dist⟩⟩ pragmatic is used distribute a message to the appropriate buffer. Figure 4.6 (right) shows the template for the ⟨⟨dist⟩⟩ pragmatic which requires a parameter which replaces `$params[0]` and is the buffer the message should be distributed to. The second line declares that the value of `$params[0]` should be available as a variable.

As an example of the generated code, the loop in the MessageBroker internal service in the Client principal is shown in Fig. 4.7. The loop is started by defining a variable, `__LOOP_VAR__`. After the `__LOOP_VAR__` is defined, the loop is entered. Finally, the template associated with the ⟨⟨endLoop⟩⟩ pragmatics has generated the code for updating `__LOOP_VAR__` according to the conditional expression given as a parameter to the ⟨⟨endLoop⟩⟩ pragmatic.

4.3 Related Implementation Technologies

PetriCode is implemented as a standalone application. However, it would have been possible to create PetriCode using another architecture. The Eclipse Mod-

```
class Client {
  ...
  def MessageBroker(){
    ...
    __LOOP_VAR__ = true
    while(__LOOP_VAR__){
      /*vars: [message:, opCode:, data:, ping:,
        pong:, pingpong:, close:, nonFinal:]*/
        inBuffer << message
      ...
      __LOOP_VAR__ = ( OPEN == true )
    }
    ...
  }
  ...
```

Figure 4.7: The generated code for the loop of the sender send service

elling Framework (EMF) [Bud04, Ecl] is a prominent technology that is often used to implement modelling software. The main advantage of EMF is that EMF provides a facility to create model editors. ePNK [Kin11] is a plug-in to EMF that allows developers to generate editors for Petri Net models. Using ePNK or EMF directly could allow us to add pragmatics as a separate label type rather than using the $\langle\langle\rangle\rangle$-notation. However, this would complicate using CPN Tools since care would have to be taken to ensure compatibility. Also, by implementing PetriCode as a standalone application it is possible to later create plug-ins for several development environments at a later stage.

The DSLs for pragmatics descriptors and template bindings are implemented by exploiting built-in features of the Groovy language. An alternative way of defining DSLs is to override the `BuilderSupport` class in the Groovy API. This approach requires implementing several abstract methods to build nodes in a user defined data-structure. While this can be a more flexible approach, based on previous experience with both approaches, it would have added unnecessary complexity to the DSLs and was not chosen for PetriCode. Using full-fledged parsers such as YACC [Joh75] and ANTLR [Par07] would have given much more flexibility in creating the syntax of the DSLs. However, this would have come with the cost of having to define the parser by hand.

We have chosen the Groovy `SimpleTemplateEngine` as the template engine and language for PetriCode. This is a powerful template language which is similar to Groovy Server Pages (GSP) [Gra]. `SimpleTemplateEngine` is also simple to integrate in PetriCode since PetriCode is implemented in the

Groovy language. In order to limit the number of dependencies of PetriCode we did not want to use template engines based on other programming languages such as PHP [AS04] or any of the large number of template engines for other programming languages. Some prominent template engines for code generation are JET [C+03] and Xpand [C+b]. These two template languages are part of the Eclipse Modelling Framework [Bud04, Ecl] (EMF). As part of EMF, both languages have requirements to the Eclipse platform and are best suited to generate code from Ecore models. Therefore, since ATTs are not created as Ecore models, these template engines were not suited for PetriCode. The same is true for implementations of the MOFM2T language [Obj08] such as Acceleo [C+a] which is designed to use models based on MOF as source models.

It would also have been possible to use model transformation approaches such as ATL [JABK08] to perform code generation. A major drawback with this approach is that Ecore, or similar, models must be created for each target language. Creating these models would likely be significantly more difficult than creating templates. Also, templates would still have to be created to generate textual code from the transformed models. Furthermore, using a model transformation approach, would require creating fairly complex transformation rules.

4.4 Summary and Contributions of Papers

The contribution of the paper *PetriCode: A Tool for Template-based Code Generation from CPN Models* [Sim14b] is to present the PetriCode tool. The PetriCode tool implements our code generation approach and is designed to be both flexible and extensible by using DSLs to specify the available pragmatics and the bindings of templates to pragmatics. PetriCode follows our approach by first parsing PA-CPNs including pragmatics and automatically deriving pragmatics. Then an ATT is constructed and, finally, the code is generated by applying the templates bound to each pragmatic of each node in the ATT and combining the generated code fragments and outputting source code files.

PetriCode complements CPN Tools by providing tool support for automatic code generation from a subclass of CPN models. Even though there exists work describing code generation based on CPN models, the CPN ecosystem has thus far lacked a tool for code generation that supports generation. PetriCode is our proposal for such a tool in the domain of protocols.

We have applied PetriCode to implement a simple framing protocol based on a PA-CPN model in [Sim14b]. In Chap. 5, we will see that PetriCode is also applicable to industrial sized protocols.

In this chapter we have concentrated the discussions on related work on implementation technologies that could have been used to implement a code generation tool. Related code generation approaches and their supporting tools have been discussed as part of Chap. 3.

Chapter 5

Evaluation of the PetriCode Code Generation Approach

In this chapter we summarize the results from evaluating our code generation approach as it is implemented in the PetriCode tool. We have evaluated our approach with regards to the requirements from presented in Chap. 1: platform independence, integrability, readability, scalability, and verifiability.

In this chapter we start by presenting the PA-CPN model of a simple framing protocol that has been used in the evaluation of several of the requirements. Then we present the evaluation of each requirement and, finally, provide a summary and discuss the contributions of the considered papers.

This chapter is based on the following papers:

- [SK14b] K.I.F. Simonsen and L.M. Kristensen. Implementing the WebSocket Protocol Based on Formal Modelling and Automated Code Generation. In *Distributed Applications and Interoperable Systems*, volume 8460 of *LNCS*, pages 104–118. Springer, 2014.
- [Sim14a] K.I.F. Simonsen. An Evaluation of Automated Code Generation with the PetriCode Approach. In *In Proc. of PNSE '14*, volume 1160 of *CEUR Workshop Proceedings*, pages 295–312. CEUR-WS.org, 2014.

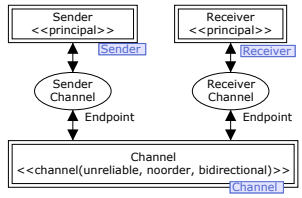


Figure 5.1: The protocol system level

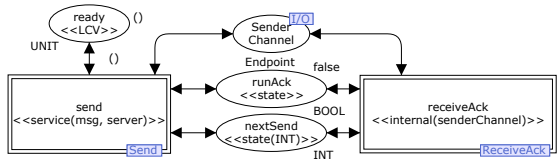


Figure 5.2: The Sender principal module

- [SKK14] K.I.F. Simonsen, L.M. Kristensen, and E. Kindler. *Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification*. DTU Compute-Technical Report-2014. Technical University of Denmark, 2014.

5.1 Example: Framing Protocol

For the evaluation of platform independence, integrability and readability we used a simple framing protocol as an example. In this section, we will briefly describe the PA-CPN model of that protocol. We used PetriCode (see Chap. 4) to generate code based on the PA-CPN model.

The framing protocol is tolerant to packet loss, reordering and allows a limited number of retransmissions. The top level of the CPN model is shown in Fig. 5.1. The model consists of three sub-modules. Sender and Receiver represent each of the principal actors of the protocol, and Channel connects the two principals.

The protocol uses sequence numbers and a flag to indicate the last frame of a message. After a frame has been sent, the receiver, if it receives the frame, sends an acknowledgement consisting of the sequence number of the frame expected next. If the acknowledgement is not received, the sender will retransmit the frame until an acknowledgement is received or the protocol fails sending the frame.

In the Sender module, shown in Fig.5.2, there are two sub-modules. The send sub-module is annotated with a ⟨⟨service⟩⟩ pragmatic and represents a service provided by this principal for sending a message. The other substitution transition receiveAck, annotated with an ⟨⟨internal⟩⟩ pragmatic, represents an internal service which is to be invoked by another service of the principal. In this example, the receiveAck service is invoked from the send service.

The Sender module also contains two places, runAck and nextSend, annotated with a ⟨⟨state⟩⟩ pragmatic which contains shared data between the two services. The ready place, annotated with a ⟨⟨LCV⟩⟩ pragmatic, is used to model the lifecycle of the Sender principal and makes sure that only a single message is sent at a time.

The send service, shown in Fig. 5.3, starts at the transition startSend which opens the channel, initialises the content of the message to be sent and the sequence number. Also, at this transition, the receiveAck internal service is started by placing a token with the colour true at the ⟨⟨state⟩⟩ place runAck. The service continues from startSend to enter a loop at the start place. Inside the loop, the sendFrame transition retrieves the next frame to be sent based on the sequence number of the frame which is matched against the sequence number incoming from the place start. The limit place is updated with the sequence number of the current frame, and the number of times the frame has been retransmitted. Then, the current frame is sent. Due to the ⟨⟨wait⟩⟩ pragmatic at the sendFrame transition, the system waits for some amount of time in order to allow acknowledgements to be received. The loop ends at place frameSent. If a token is present on the place frameSent the loop will either continue with the transition nextFrame firing or end by firing the return transition. At the return transition, state places and the channel are cleared and the service terminates.

5.2 Platform Independence

Platform independence means, in the context of our code generation approach, the ability to generate code for different platforms and programming languages. In order to demonstrate the platform independence of our approach, we showed that, from the same model, code for four different platforms could be generated: Java, Clojure, Python, and Groovy.

The code generated for each of the platforms demonstrate that our approach allows us to generate code for several platforms by providing a selection of templates for each platform. The platforms considered, spanning several popular programming paradigms, gives us confidence that our approach and tool can

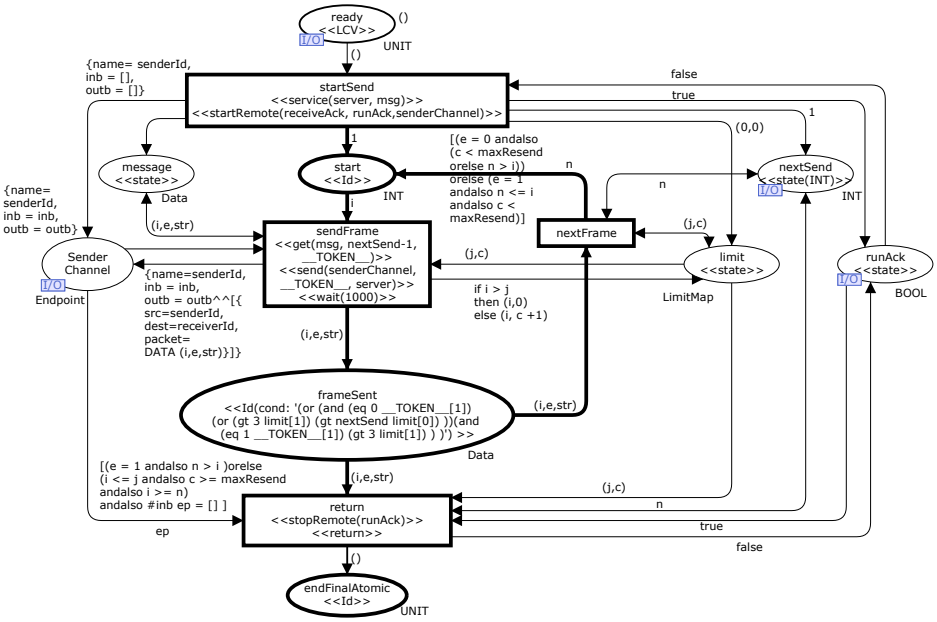


Figure 5.3: The Send service module

also be applied to generate code for many other platforms. Furthermore, we are able to generate the code for each of the platforms using the same model with the same annotations and the same code generator while only varying the code generation templates and the mappings between the pragmatics and the code generation templates.

The Groovy templates were created first and the templates for the other programming languages were based, in a varying degree, on the Groovy templates. Because of some features of the Groovy language such as optional types and native language support for lists, the templates were simple to create. For example, since Groovy is optionally typed, the Groovy templates did not have to explicitly declare the types of variables, fields and return values.

Adapting the Groovy templates to Java was, for the most part, simple since the two languages are similar in several respects. However, whereas Groovy is optionally typed, Java is statically typed and requires all variables to be typed or to be cast to specific types when accessing methods. Fulfilling Java's requirements for explicit types required functionality from PetriCode so that the templates are aware of the type of variables (see [Sim14a] for the details).

Clojure is a functional language with a different control flow from languages

such as Java. The main issue, in comparison with Groovy and Java, was related to using immutable data-structures. In Clojure all data types are, in principle, immutable. However, there is an Atom type in which values may be swapped. This was challenging because Atom values must be treated differently from pure values and lead to somewhat more verbose code than what could otherwise have been generated. Also, Clojure allows the use of Java data structures, which are mutable and thus easier to work with in this case.

Python, as Groovy, is a multi-paradigm language combining the features of several programming paradigms [Lut13]. Creating the templates for the Python code generation was, although being the only language in this survey not based on the JVM, no more difficult than for the other languages. The main challenge was to handle the significant white-spaces of the Python syntax. To support this, PetriCode contains functionality to keep track of the current indentation level (see [Sim14a] for the details).

Table 5.1 shows the sizes of the Sender and Receiver principal code (measured in code lines) for each of the platforms considered. As can be seen, the code for Python is much smaller than the others. This is due to the efficient libraries in Python and that the Python code, for technical reasons, has much fewer blank lines which is also reflected in the templates. Table 5.2 shows the sizes, in lines, for selected templates and all the templates for each platform. The sizes reported are the sizes of the actual code in the templates and generated implementations and may not correspond to the templates shown in paper, since they have been formatted in this paper for better readability. In this example, there was the same number of templates for each platform, but this is not always the case. As can be seen in Table 5.2, there is not a perfect correlation between the size of templates and the size of the generated code. This is due to, in part, some templates being more complex for some languages than others and template reuse being possible for some languages. An example is the Clojure templates, where the templates for the ⟨⟨setField⟩⟩ and ⟨⟨setValue⟩⟩ pragmatics are the same, but since the ⟨⟨setValue⟩⟩ template has more functionality than the ⟨⟨setField⟩⟩ template for all platforms, this results in a higher total number of template lines for Clojure.

To the best of our knowledge, there are no previous works where the platform independence of code generation approaches from CPNs have been evaluated. PP-CPNs [KW10] are only used to generate code for the Erlang platform while the simulation based approaches [K+04, Mor00], run on platforms where the simulation infrastructure is available.

5.3 Integrability

A requirement for our approach is that code generated by our approach can be integrated with existing software. We evaluate two types of integration: *downwards integration* and *upwards integration*. *Downwards integration* means that generated code can use different third-party libraries, and can be exemplified by having our generated code use another library for sending and receiving data from the network. *Upwards integration* means that applications can use services provided by the generated code. *Upwards integration* can be exemplified by creating a program that employs the generated protocol implementation for sending a message to a server. We have evaluated integrability in both directions using the code generated for the Java platform. However, the results are applicable to other platforms as well.

Downwards Integration. Our approach can be used to generate code for different platforms by using different templates as discussed in the previous section. The same technique was used to employ various libraries on the same platform to perform the same task. In order to change the templates that are used, we create new template bindings (see Chap. 3). The template bindings are mappings that map pragmatics to code generation templates. By varying the template bindings, developers can vary the underlying libraries used. This also means that developers must take care to use compatible template bindings to take care that templates work together since the templates can vary the types of data contained in variables.

We demonstrated downwards integrability by swapping the network library from the standard `java.net` library to Netty [Theb]. This example was chosen because networking is an important function of the network protocol domain that we consider, and because Netty is substantially different from `java.net`

Principal / Language	Groovy	Java	Clojure	Python
Sender	131	132	119	66
Receiver	81	78	68	38
Total	212	210	187	104

Table 5.1: Sizes of the generated code.

Pragmatic / Language	Groovy	Java	Clojure	Python
service	19	28	15	15
runInternal	4	10	4	3
send	9	9	8	2
All templates	154	219	251	112

Table 5.2: Size of code generation templates.

as it is an event driven library. Netty extensively uses the NIO [Hit02] IO library which leads to improved performance. This can be a very efficient way of improving the performance of protocols if needed.

Three out of twenty-one templates had to be altered to accommodate Netty as the network library. The changes were restricted to the templates that generate code for sending and receiving data from the network as well as opening the network channel. This shows that it is possible to use other, radically different, libraries than the ones originally used with reasonable effort. Thus, our code generation approach can support downwards integration even though it may be possible to create templates that makes this difficult. Furthermore, we conjecture that the template-based structure compartmentalises software fragments so that changing libraries will often result in only local changes in a small number of templates. This is because many libraries provide services for specific functions that are required to implement only a small number of pragmatics. Changing system libraries such as the java.* libraries will, of course, require changes in a larger number of templates. However, it is relatively rare to change the entire system library of a programming language in comparison to more specialised libraries.

Upwards Integration. The ability for external software to invoke the generated code is necessary for the generated code to be useful. Our approach allows this by explicitly modelling the API in the PA-CPN model in the form of principals and services which define the class and method names. To demonstrate upwards integration, we have created runners for the generated implementations for each of the platforms considered. This demonstrated that it is possible to use the generated services from third party software. It is worth noting that the explicit modelling of services in the PA-CPN model makes it simple to invoke the generated code. This result was also replicated for the WebSocket protocol in [SK14b], where we created runners to validate the generated WebSocket implementation.

To the best of our knowledge there does not exist any work evaluating the integrability of CPN based code generation approaches. It seems reasonable that approaches that do not rely on code generation templates such as simulation-based approaches, need to change the simulators or code generator in order to change the software with which it interacts. For upwards integrability it is important that there is a clearly defined interface. In [Phi06], which used a hybrid code generation approach, this is achieved by modelling the interface using UML class diagrams.

5.4 Readability

Readability of the generated code means that is can be easily read by experts in the language and platform the code is generated for. Readability is desirable for several reasons: Reviewing the generated code is facilitated by having readable code and gives confidence that the generated code behaves as expected. Furthermore, it allows developers to become confident that the templates generate the correct code individually and when they are combined. Reviewing the code can also be used to debug any code generation template that outputs flawed code. Readability also helps the integrability of the code by allowing developers using the generated code to inspect the generated code as well as the model and thereby getting an even clearer picture of the function of the code. This may give a more detailed understanding of how the generated code works.

We evaluated the readability of code generated by PetriCode in two ways. We applied a code readability metric, the *Buse-Weimer metric* [BW08], to selected snippets of the generated classes. Furthermore, we have conducted a field study where software engineers were asked to evaluate the readability of the generated code.

In order to evaluate the readability of generated code we randomly selected code snippets from the two example protocols described in Sect. 5.1 and [SKK13a]. We used code for the Java platform because the subjects of our experiments were skilled in the Java language. Also, there exist several Open Source projects from which to obtain snippets for our experiments. In addition to the generated snippets, we selected, as control group, snippets from three Open Source projects in the network protocol domain. These were the Apache FtpServer, HttpCore and Commons Net [Thea]. All three are part of the Apache project, and we consider them to be high quality projects within the network protocol domain.

We used the Buse-Weimer metric as a code readability metric. This metric was constructed by Buse and Weimer based on an experiment asking students to evaluate short code snippets with regards to readability on a scale of one to five. The experiment was used to construct the metric using machine learning methods. The mean and median score of the code snippets generated by our approach were above 0.5, indicating that the code is fairly readable. Also, the mean and median of the generated code was higher than the non-generated protocol-code. However, the generated snippets scored either very high or very low. This may indicate that further study would be advisable.

We also conducted a field study asking professional software engineers to evaluate code readability. The experiment was conducted at the JavaZone software developer conference in Oslo, Norway in September 2013. The experiment was

organised into two parts. One part evaluated the Buse-Weimer metric and another part evaluated the readability of the generated code in comparison to non-generated code. Both experiments were conducted by asking software developers to evaluate twenty small code snippets with regards to readability by assigning values, on a scale from one to five, to each code snippet. The experimental set-up was created to mimic the experiment conducted by Buse and Weimer [BW08]. The main advantage of our experiment is that the dominating majority of the participants were professional software developers instead of students. The experiment that evaluated the metric had 33 participants while the experiment evaluating the readability of generated code had 30 participants.

We believe that the empirical results are more trustworthy than the metric. While the metric evaluation showed that the generated code was more readable than the non-generated code, the empirical evaluation of the readability of the generated code indicated that the generated code is less readable than non-generated code, but still within one standard deviation. Therefore, we conclude that our generated code is less readable than hand-written code but still readable.

The importance of readability in generated code was also noted in [WH99] and several metrics, in addition to the Buse-Weimer metric, have been proposed. In [Baj11] the authors propose a code readability metric based on the layout of code. The metric is evaluated by showing that a badly formatted version of a small program scores worse than a nicely formatted version of the same program. In [PHD11] the results from the experiment performed by Buse and Weimer were used to create a simpler metric based on size and entropy that performs better than the Buse-Weimer metric in predicting human annotators. We used the Buse-Weimer metric for the evaluation of the readability of generated code because it was the only metric we were able to find an implementation of. We are unaware of any similar studies being carried out on code generated through an MDSE approach in the past.

While evaluating the readability of generated code, we were able to show using an experiment and an automated metric that code generated through our code generation approach is readable, although less readable than hand-written code. As with any empirical evaluation, this evaluation have threats to its validity. One such threat is that the chosen code, either the generated or the non-generated, is not representative for code in the protocol domain. Further threats to validity include small sample sizes and the representability of the participants (see [Sim14a] for details).

5.5 Scalability

In order to evaluate the scalability of our code generation approach we have applied our approach to the industrial sized WebSocket protocol described in Chap. 3. The complete PA-CPN model consists of 19 modules. Each of the two principals have eight sub-modules which all correspond to the external and internal services in the protocol. In total, the model consists of 136 places and 84 transitions. This reflects the complexity of the protocol, but also the high-level nature of the model which has been important in keeping the number of elements manageable. The WebSocket implementation consists of two classes containing the Client and the Server. The Client contains 839 lines and the Server 745 lines (not counting empty lines). We reused 10 templates from the library of templates provided by PetriCode. In addition, 22 new templates were added, including two templates that override existing templates (see [SK14b] and [Sim14a] for examples of templates). New templates were needed because the WebSocket protocol has many features we have not encountered with earlier examples, such as receiving and interpreting binary messages, and validating handshakes and frames.

We validated the generated WebSocket implementation in two ways. First, we created test drivers for the generated WebSocket implementation to connect to the example chat server and client [Gup] that comes with the GlassFish Application Server [Ora]. Figure 5.4 shows the chat client (upper right) and server (lower right) running together with the web-based chat client from [Gup]. The web-based client has only been modified to connect to the server using the generated API by changing a hard-coded server address. We also tested that the chat client is able to connect and communicate with a chat-server from [Gup].

The second way we validated the generated WebSocket implementation was by using the Autobahn Testsuite [Tav] version 0.5.5. The Autobahn WebSocket test-suite provides comprehensive validation of server and client implementations of the WebSocket protocol. The test-suite has been used by several high-profile projects to develop and validate WebSocket implementations including the Firefox and Jetty projects. When running the Autobahn test-suite several problems with the implementation were discovered. Most of the problems were simple oversights in the code generation templates that were easily fixed once they were identified. An example of the trivial problems that were not evident when running the chat application was that the HTTP header lines were terminated with LF instead of the mandated CRLF. However, one change to the CPN model was necessary. This was related to fragmented messages where we added a buffer for temporarily storing frames of unfinished messages and a transition to distributing non-final frames. That we were able to easily change the model to accommodate fragmented messages shows that our approach can be agile in

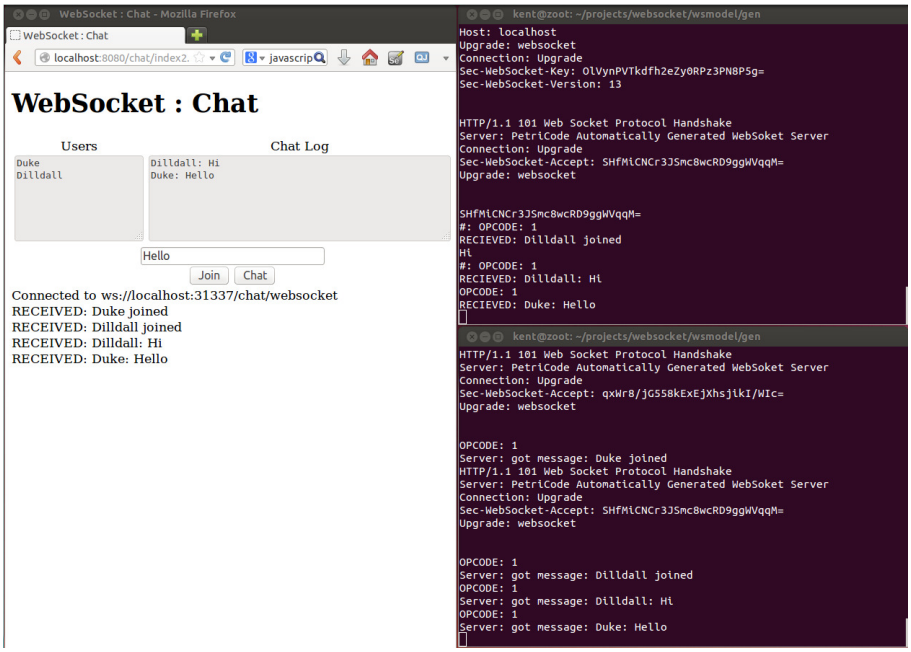


Figure 5.4: Chat server and client using the generated API (right) and a web-based chat client connected to the same server (left)

the sense of supporting incremental changes to gradually improve the model and generated implementation (see [Sim14a] for the details). The change to the model did not have any impact on the verification of the WebSocket protocol, as the verification was done after the model was validated. Both verification of models and validation of the generated code are needed to gain confidence in the correctness of a protocol implementation generated by our approach. This means that verification of models and validation of generated code may need to be re-run several times as models evolve over time either through errors being found either by validation or verification or through other means or by changes to the requirements of the protocol.

A summary of the result for the final Autobahn tests can be seen in Table 5.3. The Autobahn test suite contains 301 tests cases for the client and server. For the client, 10 test cases fail and for the server, 4 test cases fail. The extra test cases that fail on the client concern performance with large messages. We hypothesise that this has to do with differences with the handling of time-outs for the server and client testers. The test cases that fail for both the server and client are UTF-8 parser errors. This is because the Java implementation of UTF-8 parsers is more lenient than the Autobahn test-suite expects. Still, since

Table 5.3: Results for the Autobahn tests

Tests	Server Passed	Client Passed
1. Framing (text and binary messages)	16/16	16/16
2. Pings/Pongs	11/11	11/11
3. Reserved bits	7/7	7/7
4. Opcodes	10/10	10/10
5. Fragmentation	20/20	20/20
6. UTF-8 handling	137/141	137/141
7. Close handling	38/38	38/38
9. Limits/Performance	54/54	48/54
10. Auto-Fragmentation	1/1	1/1

the vast majority of the tests passed we conclude that we were able to create a correct implementation of the WebSocket protocol using our code generation approach.

5.5.1 Performance of PetriCode

We have evaluated the performance of PetriCode by using it to generate Groovy code for the three models we have used as examples in this thesis. These are two fairly simple protocols and the WebSocket protocol. Any performance differences between the two simple protocols and the WebSocket protocol should be informative as to how PetriCode scales with larger protocols. It should be noted that speed has not been a primary design goal of PetriCode, so it is possible to optimise the tool significantly with respect to speed. For this evaluation, PetriCode was changed to output timing information for each module (see Fig. 4.1).

The model of the simple framing protocol (SimpleProtocol) model was used as an example and described in [SKK13a]. This model consists of ten modules and has 45 places and 26 transitions. The model of the stop and wait protocol (SWProtocol) was used as an example and described in [SKK14]. This model consist of seven modules and has 35 places and 21 transitions. Both the SimpleProtocol and SWProtocol describe simple framing protocols. SWProtocol is slightly more complicated than the SimpleProtocol because it allows retransmissions. The WebSocket model consist of 19 modules and has 136 places and 84 transitions.

Table 5.4 shows the time spent in each of the modules in PetriCode and the total runtime of the application when generating code for the models. The total

Table 5.4: Run-time for code generation for PA-CPN models using PetriCode

Protocol	SimpleProtocol [SKK13a]	SWProtocol [SKK14]	WebSocket
Start-up	0.52s	0.51s	0.53s
Pragmatics module	6.59s	6.07s	18.20s
ATT Module	0.14s	0.12s	0.18s
Code generation module	1.77s	1.72s	3.68s
Total	9.02s	8.43s	22.59s

runtime is longer than the sum of the modules. This is because the total time includes start-up and showdown of the environment including the JVM and loading core classes in the Groovy and Java APIs. The experiment was run on an HP laptop computer with 8GB ram, an Intel i5-3210M CPU, and an SSD hard-drive.

As one could expect, the time increases with the size of the model. However, with 3.1-3.9 times the number of elements (places + transitions), the generation of the WebSocket protocol only took 2.5-2.6 times as long to run. This indicates that the approach scales fairly well although the run-time for small models is quite large. It is also notable that most of the time is used in the pragmatics module. This is because the CPN model is read and parsed in this module, which is an I/O intensive operation and the CPN model files are fairly verbose XML files. This could be improved by using more efficient methods of reading and parsing the CPN model, however, some overhead in reading and parsing large files will always be expected. Furthermore, the run-time is still acceptable for all the examples including, the industrial sized WebSocket protocol.

5.6 Verifiability

CPNs have a formal semantics which makes it possible to conduct model simulation and verification through model checking prior to code generation. This is a major advantage of an approach based on a formal modelling language as this can be used to eliminate design errors prior to code generation and testing of the generated protocol implementation. CPN Tools supports model checking of behavioural properties by means of explicit *state space exploration*. The ba-

sic idea of state space exploration is to explore all the reachable states of the model to determine whether a model satisfies a given property or not. This means that state space exploration will exhaustively explore (test) all the possible executions of the PA-CPN model. As the PA-CPN model specifies the behaviour of both the client and the server, the state space exploration exercises the client against all the possible behaviours of the server and visa versa given the configured initial state of the PA-CPN model.

We applied state space exploration of the PA-CPN model as a first test to eliminate possible errors in the model and the specification of the WebSocket protocol. For this, we adopted a lightweight approach where we considered the following behavioural properties **P0**, **P1** and **P2** of the CPN model:

P0 From the initial state it is possible to reach a state in which the WebSocket connection has been opened (i.e., both the client and the server are in the *open* state).

P1 All terminal states (i.e., states without enabled transitions) correspond to states in which the WebSocket connection has been properly closed (i.e., both the client and the server are in the *closed* state).

P2 From any reachable state, it is always possible to reach a state in which the WebSocket connection has been properly closed. This means that independently of how messages are exchanged, it is always possible to properly close the WebSocket connection.

Table 5.5 summarises the results from the verification. We have considered three possible configurations of the model. One where the client sends a message to the server; one where the server sends a message to the client; and one where both the client and the server sends one message each. The table lists the number of Nodes and Arcs in the state space, the amount of Time used to generate the state space, and the number of Terminal States. For all configurations, we were able to establish the properties P0, P1 and P2 which provides confidence in the correctness of the model. During the verification process, several minor modelling errors were identified and fixed.

Table 5.5: Results of verification of the WebSocket CPN model

Client Sends Message	Server Sends Message	Nodes	Arcs	Time (secs)	Terminal States
yes	no	2747	9,544	1	2
no	yes	2867	9,956	2	2
yes	yes	39189	177,238	246	4

The major drawback with state space exploration techniques is the state explosion problem which means that the state space in many cases grows too large to be handled with the available computing power or available memory. It is interesting to observe that the size of the state space for the model is relatively small for the configurations considered. This shows how our modelling approach makes it possible to construct models at a high-level of abstraction so that it is feasible to fully verify even industrial-sized protocols.

In [SKK14] we show that we can use the sweep-line method in order to reduce the memory consumption of the verification and thereby alleviate the state explosion problem. The basic idea of the sweep-line method is to exploit a notion of *progress* exhibited by many systems. Exploiting progress makes it possible to explore all reachable states while storing only small subsets of the state space in memory at a time. This way, much larger state spaces can be investigated since never all states need to be stored at the same time. The additional structure imposed on CPNs by PA-CPNs means that PA-CPN models have several potential sources of progress that can be exploited by the sweep-line method. The control-flow in the service modules is source of progress as there is progression from the entry point of the service towards the exit point of the service. The life-cycle of a principal is another potential source of progress as there will often be an overall intended order in which the services provided by a principal is to be invoked, and this will be reflected in the life-cycle variables of the principal.

Another source of progress are what we call *service testers*. A service tester is a module that guides the verification process. These plug into PA-CPN models in order to simulate the behaviour a user of the services provided by the protocol. A service tester is attached to a PA-CPN model through fusion sets that bind together places of the service tester and places in the service level of PA-CPN models. These places are used to control the flow of the application by invoking services and by receiving notification of the termination of services. The service tester modules are created in such a way that they inherently progress from the start of the test towards the end of the test. The sweep-line method typically requires the users to create their own progress measures for each model. However, by exploiting the structure of PA-CPNs it is possible to automatically generate such progress measures for PA-CPN models.

5.7 Summary and Contributions of Papers

The contribution *An Evaluation of Automated Code Generation with the PetriCode Approach* [Sim14a] is to evaluate our code generation approach and the

PetriCode tool. The evaluation considered the criteria of platform independence, integrability and readability of the generated code. Each of the criteria were found to be met to an acceptable degree. Another contribution of this paper was to provide evidence that the experimental results from the experiment carried out by Buse and Weimer [BW08] are relevant to professional software developers in addition to students. However, based on the discrepancy between the experimental evaluation and the metric, it seems that the metric they propose may not be applicable to code in the network protocol domain. To the best of our knowledge, there is no previous work evaluating integrability and readability of automatically generated software.

The contributions of *Implementing the WebSocket Protocol based on Formal Modelling and Automated Code Generation* [SK14b] is to evaluate our code generation approach and PetriCode with regards to scalability and verifiability. This was achieved by successfully using PetriCode to generate an implementation of the WebSocket protocol and validating the implementation. A second contribution of this paper lies in the formal analysis of the PA-CPN model of the WebSocket protocol.

The paper *A Formal Definition of Pragmatic Annotated Coloured Petri Nets for Automated Protocol Software Generation and Verification* [SKK14] discusses how we can exploit progress in PA-CPNs to make verification more efficient through the use of the sweep-line method. This paper also gives experimental results on the efficiency of various progress measures.

Platform independence, integrability and readability are evaluated using an example of a simple framing protocol. However, we argue that the results are relevant for larger protocols as well. For platform independence, larger protocols just means that more templates need to be written for each language, but the generality of the model is not affected. A similar argument can be made for integrability where the mechanisms involved, explicit API modelling at the principal level and code generation templates, are valid for large protocols. Readability is evaluated by an empirical study based on short code snippets. Therefore, the readability results should also be relevant for larger protocols. Verifiability and scalability have been evaluated using the WebSocket PA-CPN model which is an example of a large and complex protocol.

Chapter 6

Conclusions and Future Work

In this chapter we summarise the contributions and conclusions that resulted from the work done for this thesis. We also outline and discuss possible directions for future work.

6.1 Summary

In this thesis, we have presented a code generation approach where protocols implementations are automatically generated from CPN models. The approach has been implemented in the PetriCode tool and adds code generation capabilities to the CPN ecosystem for the protocol domain.

Our code generation approach relies on a new class of CPNs called Pragmatics Annotated Coloured Petri Nets (PA-CPNs). PA-CPNs are designed to be descriptive while also facilitating code generation. This was realised by enforcing a hierarchical structure where the first level defines the principal agents of the protocol, the second level, defines the available services, and the third level defines the detailed behaviour of each service.

In PA-CPNs, the model elements are annotated by code generation pragmatics. The pragmatics allow us to give additional information to the code generator. PA-CPNs and pragmatics are independent of the target platforms. Platform specific information are added by code generation templates. Pragmatics are bound to templates by template bindings. To allow flexibility of the protocols that can be generated and the target platforms, PetriCode allows users to define pragmatics and template bindings using DSLs. All information about the target platform are contained in code generation templates. Thus, platforms are chosen by specifying a set of template bindings.

Our code generation approach consist of three steps. The first step is to parse a PA-CPN model and deriving pragmatics. The next step is to generate an Abstract Template Tree (ATT). Finally, the ATT is transformed to code by executing the templates associated with each pragmatic on each node of the ATT and combine the code fragments which contain all the platform specific information.

Our code generation approach is implemented by the PetriCode tool. PetriCode is designed to be extensible by using DSLs to allow the users to easily change and expand the available pragmatics and template bindings through template and binding descriptors.

We have evaluated our code generation approach based on the criteria we described in Chap. 1:

- **Readability.** We evaluated the readability of generated code using an automatic metric and an empirical study. In both the metric and experimental evaluations we used randomly selected snippets of generated code and compared them to randomly selected snippets from high-quality projects in the protocol domain. The metric based evaluation showed that the generated code was somewhat more readable than non-generated code while the empirical study showed that the generated was less readable but within a standard deviation of the non-generated code.
- **Platform Independence.** Platform independence was evaluated by using PetriCode to generate implementations of a protocol for four different platforms based on the same PA-CPN model annotated with the same pragmatics. This evaluation also demonstrated that PA-CPN model are general and sufficiently complete to generate code for several platforms without modification.
- **Integrability.** We demonstrated integrability of code generated by PetriCode by showing how our approach allows generated code to use different underlying libraries. We also used an example program to demonstrate

that third party applications can use our applications through the interface defined in the principal layer of PA-CPN models. This part of the evaluation of integrability was also repeated with the generated WebSocket implementation.

- **Scalability.** We evaluated scalability by using PetriCode to automatically generate an implementation of the WebSocket protocol based on a PA-CPN model. The WebSocket protocols is an industrial sized protocol and generating code for it shows that our approach is able to scale to industrial sized protocols. The generated WebSocket implementation was validated using an example chat application and by applying a comprehensive test suite.
- **Verifiability.** Verifiability was evaluated by verifying connection establishment and termination properties of the WebSocket PA-CPN model using state space exploration. The verification showed that PA-CPNs, while being descriptive and code generation models, are also verifiable. Furthermore, we have described how we can exploit the structure of PA-CPNs with the sweep-line method to allow for space efficient verification of PA-CPN models.

6.2 Conclusions

PA-CPNs are inspired by the survey of the use of CPNs for the modelling and verification of protocols discussed in Chap. 2. We argue that PA-CPNs are descriptive since PA-CPNs have a fairly simple but still strict structure. Furthermore, the service level modules have an imperative structure with a clearly marked control-flow via the use of $\langle\langle \mathsf{Id} \rangle\rangle$ pragmatics. We have also shown that PA-CPNs are verifiable and scalable while allowing code generation by being able to model and verify the WebSocket protocol using PA-CPNs. Since PA-CPN models are usable as descriptive, verification and code generation models, this removes the need to keep different models synchronised, which otherwise would be challenging.

We have constructed a code generation method that translates PA-CPN models to code for various platforms. We have also developed a prototype tool that follows this approach. Our approach is flexible with respect to the pragmatics and templates that are available. In the PetriCode tool, pragmatics and template bindings can be defined using DSLs. This allows us to extend the code generation approach and tool to fit the needs of the user in different circumstances.

We have been able to evaluate all the key requirement we set out to achieve

with our code generation approach. All the evaluations have been positive, in that they indicate that our requirements are met. We have shown by example that our approach is scales to industrial sized protocol such as the WebSocket protocol. Furthermore, there is no reason to believe that we could not generate code for protocols that are significantly larger than the WebSocket protocols as well.

We evaluated verifiability of PA-CPNs and the scalability of our approach based on a study by generating code for the WebSocket protocol. This study showed that, even though the PA-CPN model of the WebSocket protocol is fairly large, the state space remains manageable. This, together with the results showing that PA-CPNs are amenable to the sweep-line method for state space exploration shows that PA-CPNs are verifiable to a reasonable degree. The code generation of the WebSocket protocol also showed that our approach is able to generate code for industrial sized protocols and that it can be done within a short time.

We showed that platform independence is supported by our approach. We showed this by generating code for several languages using the same PA-CPN model. The breadth of the platforms we had chosen gives us confidence that the platform independence is extendable to other platforms. However, we have only evaluated one programming language that does not run on the JVM platform.

Integrability was evaluated by showing that we are able to use the generated code for several platforms by writing programs that use the services of the generated protocol. We were also able to show that our approach allowed us to change underlying libraries. We demonstrated this by changing the network library and showing that even this, fairly extensive change, only required changes local to a few templates that had to do with networking. Even though this evaluation is somewhat crude, the ease of developing the runner programs and the fact that the network API is a central part of protocol implementation gives us confidence that our result evaluating integrability has merit.

Readability was evaluated by using an automated metric and an empirical study. The metric indicated that the generated code was, on average slightly more readable than the generated code, with a large amount of variability in the scores. In the empirical study we asked developers to evaluate the readability of generated and handwritten code. In this study the generated code was evaluated to be somewhat less readable than generated code, but within the standard deviation.

The main finding of this thesis is that have been able to create a code generation approach that generates high quality code for communication protocols based on CPN models. The PA-CPN models that we use as a basis for code generation are also descriptive and amenable to verification. Furthermore, we have showed

that our approach scales to industrial sized protocols, is platform independent and that the generated code is integrable and readable.

Table 6.1 summarises the extent to which related code generation approaches meet the requirements we have evaluated for PetriCode. The approach presented in [Phi06] is a hybrid of simulation based and structural analysis approaches to code generation for HLPNs. The paper argues that the hybrid approach produces more readable code than a naive simulation approach because fewer checks are needed in the code. The approach, as described in [Phi06] only produces code for the Java platform, however, it is possible that the approach could be adapted for other platforms. The paper does not comment on the verifiability of the models. However, it seems plausible that the Petri Nets models in the approach are verifiable. We consider this approach to produce interoperable code because the API of the generated code is defined by UML class diagrams. The paper gives little evidence that the approach scales to large applications.

In [LT07] the authors claim to generate readable by creating code with constructs that are similar to what human programmers would have created. The approach is tailored to the Java programming language and no evidence is given that the approach is transferable to other platforms. WF-nets, which are similar to the models that are used as an intermediary step in the code generation, are known to be verifiable [Aal97]. The paper does not argue that the code generated is integrable or that the approach scales well to larger systems although the paper mentions that there exists efficient algorithms for translating the models to code.

In [K+04] the authors make no mention on the readability or the platform independence of the generated code. Moreover, the paper clearly states that RENEW is based on the Java platform. Although the paper does not discuss the verifiability of the models used, the reference net models that are used RENEW are known to be verifiable [MWW10]. The paper does not mention whether the generated code is integrable at the code level with third-party code. However, the paper says that reasonably large applications, around 100 classes, can be created using this approach which indicates that it is scalable.

The approach presented in [Mor00] is a simulation based approach. The paper does not make any claims about the readability of the code. It is also dependent on the SML platform. Since the approach extracts SML code from the model, it is possible to use verifiable models. The paper also does not make any claims about the integrability of the generated code. The paper mentions that the generated code becomes very large even for moderately sized models, so the approach is likely not scalable in the form presented in [Mor00].

Approach	Readable	Platform independence	Verifiable	Integrable	Scalable
PetriCode	✓	✓	✓	✓	✓
[Phi06]	✓	✗	✓	✓	✗
[LT07]	✓	✗	✓	✗	✗
Renew [K+04]	✗	✗	✓	✗	✓
[Mor00]	✗	✗	✓	✗	✗
[KMZ+08]	✗	✗	✓	✗	✓
PP-CPNs [KW10]	✗	✗	✓	✗	✗
[vdAJL05]	✗	✗	✗	✗	✗

Table 6.1: Properties of related Petri net based code generation approaches.

The approach presented in [KMZ+08] is similar to the approach in [Mor00]. The paper does not make any claims about the readability of the code and is also based on the SML platform. The paper discusses state space exploration and concludes that this is feasible. The paper makes no claims about the integrability of the generated code. The paper also does not mention the scalability of the approach, however, the size of the example is large, so we assume that it is scalable.

In [KW10] PP-CPNs are used as the basis for code generation targeting the Erlang language. The papers makes no claims about the readability of the generated code. Furthermore, the approach is tailored to the Erlang platform and may not be easily adapted for to other platforms even though PP-CPNs and the intermediary representation CFGs are independent of the target language. PP-CPN models are verifiable using state space exploration. However, the paper make no claims about the integrability or the scalability of the approach.

In [vdAJL05], code for BPEL is generated. The approach is targeted at BPEL and does not create code for other languages. The papers also makes no claims on the verifiability, integrability, readability, or scalability of the approach.

We based our evaluation of the related approaches on best effort reading of the relevant papers. This was due to the fact that, in many cases, the supporting computer tools are not publicly available. As can be seen from the table, PetriCode is the only one that meets all the requirements we have set for our approach.

6.3 Future Work

We have identified several directions for future work based on the work in this thesis that will allow our approach and tool to mature further and increase in scope.

In this thesis we have applied our approach to two simple protocols and one industrial sized protocol. In order to further evaluate the applicability of our approach, studies of further use cases in the protocol and adjacent domains would be needed.

Possible future improvements to the PetriCode tool include support for automatically generating progress measures for verification purposes. This would make it easier for users to employ the sweep-line method for verifying larger protocols using PA-CPNs. It would also be interesting to investigate how suitable PA-CPNs are to be used with various other advanced verification techniques.

We have evaluated the readability of the generated code, however, it is also important that PA-CPN models are readable. Model readability could be evaluated by an experiment similar to the experiment we used to evaluate code readability. Some challenges with evaluating the readability of PA-CPNs is that experts on CPN models are more scarce than experts in the Java language. Furthermore, the availability of descriptive CPN models to use as comparison models is also scarce since most work with CPN until now has been more focused on verifiability than descriptiveness. This makes such an evaluation challenging. However, one possible approach would be to conduct interviews with protocol and CPN experts to evaluate the readability of PA-CPNs.

In order to further improve the readability of code generated by our PetriCode, PetriCode could be extended to allow the results from certain templates to be factored out into methods or functions. Another improvement would be to support for easily combining templates so that users can make complex templates by combining several simple and easy to inspect templates. Being able to hide and display pragmatics and verification artefacts such as test-drivers in a CPN editor could also promote the descriptiveness of the model. However, this is currently out of the scope of PetriCode.

Another possible direction for future work is to add support for validation of code generation templates. One idea for doing this would be to use automated testing of the templates. This could be done either by statically checking the text or by checking that the produced code runs as expected by using unit testing and similar techniques. It would also be interesting to investigate the possibility of using the PA-CPN models and state spaces to automatically generate test-

cases for the protocol implementation. Finally, formal verification of the code generation process and the generated code would be ideal, although it might be difficult to achieve in practice.

While we have provided a preliminary evaluation of the speed of code generation, evaluating the speed of generated code has been left for future work. It is possible that the structure imposed by PetriCode may introduce some overhead in the generated code. However, our hypothesis is that it is possible, through careful design of pragmatics and templates, to generate protocols with good performance characteristics for most platforms using PetriCode.

Evaluating the quality of automatically generated code in terms of the correctness and the amount of flaws in generated code compared to non-generated code is left for future work. Such qualities are usually evaluated based on production code. Since PetriCode has not been used to generate production software yet, we have not been able to evaluate these qualities. However, in [Tol04] the authors argue that template based approaches can yield better quality code since the templates can be written by experts and thoroughly checked, while the models are validated and therefore also has a high degree of correctness and few flaws. We agree with this argument although we acknowledge that, to the best of our knowledge, there is not yet much strong empirical evidence supporting this argument. Evaluating maintainability, is another potential avenue of further evaluation.

A final direction for future work is to explore the use of our code generation approach for other domains than communication protocols. We have started this by using PetriCode for generating code for the embedded domain [KS14]. This work can be expanded by using our approach to generate code for Web services and Web applications. These domains are good candidates for our approach since they are close to the protocol domain even though several approaches are able to generate code for these domains already. Furthermore, this could allow us to combine PetriCode with existing code generation approaches for web applications.

In this thesis we have described our code generation approach and the PetriCode tool. Furthermore, we have shown that the PetriCode tool is able to generate quality code for the network protocol domain. We believe that this approach and even the PetriCode tool, with suitable modifications, can be generalised to be usable for most programming domains.

Bibliography

[15401] ISO/IEC 15437. Information technology. Enhancements to LOTOS (E-LOTOS), September 2001.

[3GP07] 3GPP. Digital Cellular Telecommunications System (Phase 2+); Generic Access to the A/Gb Interface; Stage 2. 3GPP TS 43.318 version 6.9.0 Release 6, March 2007.

[Aal97] W.M.P. Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer, 1997.

[ALPT05] M. Alanen, J. Lilius, I. Porres, and D. Truscan. *On Modeling Techniques for Supporting Model Driven Development of Protocol Processing Applications*, pages 305–328. Springer, 2005.

[AS04] L. Atkinson and Z. Suraski. *Core PHP programming.* Prentice Hall Professional, 2004.

[Baj11] A.O. Bajeh. A Novel Metric For Measuring The Readability of Software Source Code. *Journal of Emerging Trends in Computing and Information Sciences*, 2(10):492–497, 2011.

[BB87] T Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14:25–59, 1987.

[BBSCdlF05] M. Encarnación Beato, Manuel Barrio-Solórzano, Carlos E. Cuesta, and Pablo de la Fuente. {UML} automatic verification

tool with formal methods. *Electronic Notes in Theoretical Computer Science*, 127(4):3 – 16, 2005. Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM 2004) Visual Languages and Formal Methods 2004.

[BCW12] M. Brambilla, J. Cabot, and M. Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool, 2012.

[BD02] F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *Communications Surveys Tutorials, IEEE*, 4(1):2–20, 2002.

[BGH04a] J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 210–290. Springer, 2004.

[BGH04b] J. Billington, G.E. Gallasch, and B. Han. A coloured Petri net approach to protocol verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, 2004.

[BH07] J. Billington and B. Han. Modelling and Analysing the Functional Behaviour of TCPs Connection Management Procedures. *International Journal on Software Tools for Technology Transfer*, 9(3-4):269–304, 2007.

[BK08a] C. Baier and J-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BK08b] C. Baier and J-P Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Boc78] G. Bochmann. Finite state description of protocols. *Computer Networks*, pages 361–372, 1978.

[Bud04] F. Budinsky. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004.

[BVA08] J. Billington and S. Vanit-Anunchai. Coloured Petri Net Modelling of an Evolving Internet Protocol Standard: The Datagram Congestion Control Protocol. *Fundamenta Informaticae*, 88(3):357–385, 2008.

[BW08] R.P.L. Buse and W.R. Weimer. A metric for software readability. In *Proc. of ISSTA'08*, pages 121–130, NY, USA, 2008. ACM.

[BY09] J. Billington and C. Yuan. On Modelling and Analysing the Dynamic MANET On-Demand (DYMO) Routing Protocol. In *Transactions on Petri Nets and Other Models of Concurrency*, volume 5800 of *LNCS*, pages 98–126, 2009.

[C+a] Eclipse Consortium et al. Acceleo. http://www.eclipse.org/modeling/m2t/?project=xpand.

[C+b] Eclipse Consortium et al. Xpand. http://www.eclipse.org/acceleo/.

[C+03] Eclipse Consortium et al. Java emitter templates (jet), 2003. http://www.eclipse.org/modeling/m2t/?project=jet.

[CDE+10] C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification. In *Proc. of ICATPN'10*, volume 6128 of *LNCS*, pages 145–164. Springer, 2010.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. *Formal Methods in System Design*, 9:77–104, 1996.

[CKM01] S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.

[Cou87] J.P. Courtiat. Petri Nets Based Semantics for Estelle. Technical Report SEDOS/109, SEDOS, nov. 1987.

[CP07a] I.D. Chakeres and C.E. Perkins. Dynamic MANET On-demand (DYMO) Routing. http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-10.txt, July 2007. Internet-Draft. Work in Progress.

[CP07b] I.D. Chakeres and C.E. Perkins. Dynamic MANET On-demand (DYMO) Routing. http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-11.txt, November 2007. Internet-Draft. Work in Progress.

[Cro69] Crocker, S. Documentation Conventions, April 1969. RFC 3.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPV6) Specification, December 1998. RFC 2460.

[DL08] L.G. Ding and L. Liu. Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets. In *Proc. of ICATPN'08*, volume 5062 of *LNCS*, pages 132–151. Springer, 2008.

[DPRZ12] E. Domínguez, B. Pérez, Á.L. Rubio, and M.A. Zapata. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*, 54(10):1045 – 1066, 2012.

[Ecl] Eclipse Modeling Framework. *Project Web Site*. http://www.eclipse.org/emf/.

[EKK03] W. El Kaim and F. Kordon. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*, chapter Code Generation. Springer, 2003.

[EKK08] K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *Proc. of ICATPN'08*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, June 1999. RFC 2616.

[FK09] P. Fleischer and L.M. Kristensen. Modelling and Validation of Secure Connection Establishment in a Generic Access Network Scenario. *Fundamenta Informaticae*, 94(3-4):361–386, 2009.

[FM11] I. Fette and A. Melnikov. The websocket protocol, December 2011. RFC 6455.

[GB00] S. Gordon and J. Billington. Analysing the WAP Class 2 Wireless Transaction Protocol Using Coloured Petri Nets. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 207–226. Springer, 2000.

[GB05] G. E. Gallasch and J. Billington. Using Parametric Automata for the Verification of the Stop and Wait Class of Protocols. In *Proc 3rd Int. Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *LNCS*, pages 457–473. Springer, 2005.

[GB06] G. E. Gallasch and J. Billington. A Parametric State Space for the Analysis of the Infinite Class of Stop-and-Wait Protocols. In *Proc. of SPIN Workshop on Model Checking of Software*, volume 3925 of *LNCS*, pages 201–218. Springer, 2006.

[GBVAK07] G. E. Gallasch, J. Billington, S. Vanit-Anunchai, and L.M. Kristensen. Checking Safety Properties On-the-fly with the Sweep-Line Method. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):371–392, 2007.

[GH06] V. Gehlot and A. Hayrapetyan. A CPN Model of a SIP-Based Dynamic Discovery Protocol for Webservices in a Mobile Environment. In *Proc. of 7th Workshop and Tutorial onPractical Use of Coloured Petri Nets and the CPN Tools (CPN'06)*, 2006.

[GH07] V. Gehlot and A. Hayrapetyan. A Formalized and Validated Executable Model of the SIP-based Presence Protocol for Mobile Applications. In *Proceedings of the 45th Annual ACM Southeast Regional Conference*, pages 185–190. ACM, 2007.

[GHB05] G.E. Gallasch, B. Han, and J. Billington. Sweep-Line Analysis of TCP Connection Management. In *ICFEM*, volume 3785 of *LNCS*, pages 156–172. Springer, 2005.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GKB02] S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer, 2002.

[GOBK04] G.E. Gallasch, C. Ouyang, J. Billington, and L.M. Kristensen. Experimenting with Progress Mappings for the Sweep-Line Analysis of the Internet Open Trading Protocol. In *Proc. of 5th Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools (CPN'04)*, pages 19–38, 2004.

[Gor08] S. Gordon. Formal Analysis of PANA Authentication and Authorisation Protocol. In *Proc. of 9th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 277–284. IEEE Computer Society, 2008.

[Gra] Grails Documentation. *Groovy Server Pages*. http://grails.org/doc/2.2.x/guide/theWebLayer.html.

[Gri06] P. Grimstrup. Interworking Description for IKEv2 Library. Ericsson Internal. Document No. 155 10-FCP 101 4328 Uen, September 2006.

[Groa] Groovy. *Project Web Site.* http://groovy.codehaus.org.

[Grob] O. M. G. Group. *UML Specification, Version 2.0.*

[Gup] Gupta, A. *Chat Sever using WebSocket in GlassFish 4.* https://blogs.oracle.com/arungupta/entry/chat_sever_using_websocket_totd.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hit02] R. Hitchens. *Java NIO.* O'Reilly Media, Inc., 2002.

[Hol91] G.J. Holzmann. *DESIGN AND VALIDATION OF COMPUTER PROTOCOLS.* PRENTICE-HALL, 1991.

[Hol97] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, May 1997.

[I] The Internet Engineering Task Force (IETF). http://www.ietf.org.

[ISO] ISO9074. Information Processing Systems - Open Systems Interconnection: ESTELLE (FOrmal Description Technique Based on an Extended State Transition Model.

[ISO89] ISO89 ISO/IEC. Information Processing Systems - Open Systems Interconnection: LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, February 1989. IS 8807.

[IT99] ITU-T. Recommendation z.100 (11/99) specification and description language (sdl), 1999.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[Jen94] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2: Analysis Methods.* Monographs in Theoretical Computer Science. Springer, 1994.

[JK99] J.B. Jørgensen and L.M. Kristensen. Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):714–732, 1999.

[JK03] J.B. Jørgensen and L.M. Kristensen. Verification of Coloured Petri Nets Using State Spaces with Equivalence Classes. In *Petri Net Approaches for Modelling and Validation*, volume 1 of *LINCOM Studies in Computer Science*, chapter 2, pages 17–34. Lincoln Europa, 2003.

[JKM12] K. Jensen, L.M. Kristensen, and T. Mailund. The sweep-line state space exploration method. *Theoretical Computer Science*, 429:169–179, 2012.

[JKW07] K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.

[Joh75] S.C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[K+04] O. Kummer et al. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In *Proc. of ICATPN'04*, volume 3099 of *LNCS*, pages 484–493. Springer, 2004.

[KHHH04] P. Kukkala, V. Helminen, M. Hannikainen, and T.D. Hamalainen. UML 2.0 implementation of an embedded WLAN protocol. In *Proc. of PIMRC '04*, volume 2, pages 1158–1162 Vol.2, 2004.

[Kin11] E. Kindler. The ePNK: an extensible petri net tool for PNML. In *Applications and Theory of Petri Nets*, pages 318–327. Springer, 2011.

[KJ04] L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Prot ocol for Mobile Ad-hoc Networks. In *Proc. of Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.

[KMZ+08] L. M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G. E. Gallasch. Model-based development of a course of action scheduling tool. *International Journal on Software Tools for Technology Transfer*, 10:5–14, 2008.

[Kri10] L.M. Kristensen. A Perspective on Explicit State Space Exploration of Coloured Petri Nets: Past, Present, and Future. In *Proc. of ICATPN'10*, volume 6128 of *LNCS*, pages 39–42. Springer, 2010.

[KS13] L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *ToPNoc VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.

[KS14] S.A. Kumar and K.I.F. Simonsen. Towards a Model-Based Development Approach for Wireless Sensor-Actuator Network Protocols. In *Proc. of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, pages 35–39. ACM, 2014.

[KV98] L.M. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In *Proc. of ICATPN'98*, volume 1420 of *LNCS*, pages 104–123. Springer, 1998.

[KW10] L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.

[KWN05] L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Adhoc Networks. In *Proc. of Fifth International Conference on Integrated Formal Methods (IFM'05)*, volume 3771 of *LNCS*, pages 266–286. Springer, 2005.

[Lak09] C. Lakos. Modelling Mobile IP with Mobile Petri Nets. *Transactions on Petri Nets and Other Models of Concurrency*, 5800:127–158, 2009.

[LB07] L. Liu and J. Billington. Verification of the Capability Exchange Signalling protocol. *STTT*, 9(3-4):305–326, 2007.

[Liu09] L. Liu. Verification of the SIP Transaction Using Coloured Petri Nets. In Bernard Mans, editor, *Thirty-Second Australasian Computer Science Conference (ACSC 2009)*, volume 91 of *CRPIT*, pages 63–72, Wellington, New Zealand, 2009. ACS.

[Liu10] L. Liu. Security Analysis of Session Initiation Protocol - A Methodology Based on Coloured Petri Nets. In *Proc. of the 2010 International Cyber Resilience Conference*, 2010.

[LK00] L. Lorentsen and L.M. Kristensen. Modelling and Analysis of a Danfoss Flowmeter System Using Coloured Petri Nets. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 346–366. Springer, 2000.

[LP99] J. Lilius and I.P. Paltor. vuml: a tool for verifying uml models. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 255–258, Oct 1999.

[LT07] K. B. Lassen and S. Tjell. Translating colored control flow nets into readable Java via annotated Java workflow nets. In *Proc. of 8th CPN Workshop*, 2007.

[Lut13] M. Lutz. *Learning python.* "O'Reilly Media, Inc.", 2013.

[MI02] Y. Matsumoto and K Ishituka. Ruby programming language, 2002.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Mor00] K. H. Mortensen. Automatic code generation method based on coloured Petri net models applied on an access control system. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 367–386, 2000.

[MWW10] M. Mascheroni, T. Wagner, and L. Wüstenberg. Verifying Reference Nets by Means of Hypernets: A Plugin for Renew. In *Proc. of the International Workshop on Petri Nets and Software Engineering, PNSE*, Berichte des Fachbereichs Informatik der Universität Hamburg, pages 39–54. Universität Hamburg, 2010.

[NES+11] Charles O Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java.* Pragmatic Bookshelf, 2011.

[O+08] Martin Odersky et al. The scala programming language. *URL http://www. scala-lang. org*, 2008.

[OB03] C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proc. of 4th International EC-Web Conference*, volume 2738 of *LNCS*, pages 292–302. Springer, 2003.

[OB04] C. Ouyang and J. Billington. Formal Analysis of the Internet Open Trading Protocol. In *Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops*, volume 3236 of *LNCS*, pages 1–15. Springer, 2004.

[Obj08] Object Management Group. *MOF Model to Text Transformation Language (MOFM2T)*, January 2008. http://www.omg.org/spec/MOFM2T/1.0/.

[OKB02a] C. Ouyang, L.M. Kristensen, and J. Billington. A Formal and Executable Specification of the Internet Open Trading Protocol. In *Proc. of Third International Conference on E-Commerce and Web Technologies*, volume 2455 of *LNCS*, pages 377–387. Springer, 2002.

[OKB02b] C. Ouyang, L.M. Kristensen, and J. Billington. A Formal Service Specification for the Internet Open Trading Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 352–373. Springer, 2002.

[OMG] OMG Model Driven Architecture. *Web Site*. http://www.omg.org/mda/.

[Ora] Oracle Corporation. *GlassFish Application Server*. https://glassfish.java.net/.

[Par07] Terence Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf, 2007.

[PdSD+00] C.L. Pereira, Jr. da Silva, D.C., R.G. Duarte, A.O. Fernandes, L.H. Canaan, C.J.N. Coelho, and L.L. Ambrosio. Jade: An embedded systems specification, code generation and optimization tool. In *Proc. SBCCI '00*, pages 263–268, 2000.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[Pet77] C.A. Petri. Communication disciplines. In *Proceedings of the Joint IBM University of Newcastle upon Tyne seminar*, pages 171–183, 1977.

[Pet13] M. Petre. UML in practice. In *Proc. of the 2013 International Conference on Software Engineering (ICSE'13)*, pages 722–731. IEEE Press, 2013.

[PHD11] D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proceedings of MSR '11*, pages 73–82. ACM, 2011.

[Phi06] S. Philippi. Automatic code generation from high-level Petri-nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444 – 1455, 2006.

[Pos81] J. Postel. Transmission control protocol, September 1981. RFC 793.

[Pow04] A Powell. Generate code with eclipse's java emitter templates. *IBM Developerworks*, 2004.

[PvKHT00] J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. UML for protocol engineering-extensions and experiences. In *Proc. of TOOLS '00*, pages 82–93, 2000.

[Rei85] W. Reisig. *Petri Nets - An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[RWL+03] A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ICATPN'03*, volume 2679 of *LNCS*, pages 450–462. Springer, 2003. http://www.cpntools.org.

[Sim11] K.I.F. Simonsen. On the use of Pragmatics for Model-based Development of Protocol Software. In *Proc of PNSE '11*, volume 723 of *CEUR Workshop Proceedings*, pages 179–190. CEUR-WS.org, 2011.

[Sim14a] K.I.F. Simonsen. An Evaluation of Automated Code Generation with the PetriCode Approach. In *In Proc. of PNSE '14*, volume 1160 of *CEUR Workshop Proceedings*, pages 295–312. CEUR-WS.org, 2014.

[Sim14b] K.I.F. Simonsen. PetriCode: a tool for template-based code generation from CPN models. In *Software Engineering and Formal Methods*, pages 151–163. Springer, 2014.

[SK12] K.I.F. Simonsen and L.M. Kristensen. Towards a CPN-based Modelling Approach for Reconciling Verification and Implementation of Protocol Models. In *Proc. of MOMPES'12*, volume 7706 of *LNCS*, pages 106–125. Springer, 2012.

[SK14a] K.I.F. Simonsen and L.M. Kristensen. A Pragmatic Approach for Transforming Coloured Petri Net Models Into Code: A Case Study of the IETF WebSocket Protocol. In *Abstracts of the 1st International Joint Symposium on Program and Model Transformations*, 2014.

[SK14b] K.I.F. Simonsen and L.M. Kristensen. Implementing the WebSocket Protocol Based on Formal Modelling and Automated Code Generation. In *Distributed Applications and Interoperable Systems*, volume 8460 of *LNCS*, pages 104–118. Springer, 2014.

[SKK12] K.I.F. Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocols from CPN models Annotated with Pragmatics – Extended Abstract. In *Proc of* 24^{th} *Nordic Workshop on Programming Theory*. NWPT, November 2012.

[SKK13a] K.I.F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.

[SKK13b] K.I.F Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocol Software from CPN models Annotated with Pragmatics. Technical Report IMM-Technical Reports-2013-01, Technical University of Denmark, DTU Informatics, January 2013. Available via http://bit.ly/WwH2hf.

[SKK14] K.I.F. Simonsen, L.M. Kristensen, and E. Kindler. *Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification*. DTU Compute-Technical Report-2014. Technical University of Denmark, 2014.

[SMR10] K.I.F. Simonsen, A. Mantz, F. Rossini, and A. Rutle. Groovy and Grails meets Eclipse Modelling Framework. *Norsk informatikkonferanse (NIK)*, pages 34–43, 2010.

[SOSF09] S. Suriadi, C. Ouyang, J. Smith, and E. Foo. Modeling and Verification of Privacy Enhancing Protocols. In *Proc. of ICFEM'09*, volume 5885 of *LNCS*, pages 127–146. Springer, 2009.

[Tav] Tavendo GmbH. *Autobahn|Testsuite*. http://autobahn.ws/testsuite/.

[Thea] The Apache Software Foundation. FtpServer http://mina.apache.org/ftpserver-project/, HttpCore https://hc.apache.org/, Commons Net http://commons.apache.org/proper/commons-net/.

[Theb] The Netty project. *Netty*. http://netty.io.

[Tol04] J.P. Tolvanen. Metaedit+: domain-specific modeling for full code generation demonstrated [gpce]. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 39–40. ACM, 2004.

[Ull98] J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.

[VA08] S. Vanit-Anunchai. Towards Formal Modelling and Analysis of SCTP Connection Management. In *Proc. of CPN'09*, pages 163–182, 2008.

[VABG08] S. Vanit-Anunchai, J. Billington, and G. E. Gallasch. Analysis of the Datagram Congestion Control Protocols Connection Management Procedures Using the Sweep-line Method. *International Journal on Software Tools for Technology Transfer*, 10(1):29–56, 2008.

[Val91] A. Valmari. Stubborn Sets of Coloured Petri Nets. In *Proc. of ICATPN'91*, pages 102–121, 1991.

[Val98] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.

[VB03a] M.E. Villapol and J. Billington. A Coloured Petri Net Approach to Formalising and Analysing the Resource Reservation Protocol. *CLEI Electron. J.*, 6(1), 2003.

[VB03b] M.E. Villapol and J. Billington. Analysing Properties of the Resource Reservation Protocol. In *Proc. of ICATPN'03*, volume 2679 of *LNCS*, pages 377–396. Springer, 2003.

[vdAJL05] W.M.B van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System. In *In Proc. of CoopIS'05*, volume 3760 of *LNCS*, pages 22–39. Springer, 2005.

[WH99] M.W. Whalen and M.P.E. Heimdahl. On the requirements of high-integrity code generation. In *The 4th IEEE International Symposium on High-Assurance Systems Engineering*, HASE '99, pages 217–, Washington, DC, USA, 1999. IEEE Computer Society.

[WK09] M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In *Proc. of ICATPN '09*, volume 5606 of *LNCS*, pages 313–322. Springer, 2009.

[WL06] M. Westergaard and K. B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ICATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer, 2006.

Part II

Papers

Chapter 7

Applications of Coloured Petri Nets for Functional Validation of Protocol Designs

Applications of Coloured Petri Nets for Functional Validation of Protocol Designs

Lars M. Kristensen[1⋆] and Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computer Engineering, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
[2] DTU Informatics, Technical University of Denmark, Denmark
Email: kisi@imm.dtu.dk

Abstract. Communication protocols constitute central building blocks in most modern IT systems as they define components, rules, and languages that make data communication possible. The development of correct protocols is a challenging engineering discipline, making modelling and validation of protocol design an important application domain for Coloured Petri Nets (CPNs). We illustrate the practical application of CPNs for protocol validation by focusing on selected aspects of four recent projects involving industrial-sized protocols. These projects demonstrate how CPNs can be used to model protocol elements and improve protocol specifications, how state space exploration can be used to verify protocol properties, and how behavioural visualisation in combination with a CPN model provides an effective way of rapidly constructing an executable prototype of a protocol design.

1 Introduction

Communication protocols play an important role in most IT systems. A prominent example is the vast amount of web applications that are in use today for, e.g., online banking, shopping, government administration, and entertainment. The services provided by these applications all rely on the protocols governing the operation of the Internet. Other examples are telecommunication systems, logistic systems with sensors and actuators, and control systems in vehicles. All these systems rely heavily on communication and synchronisation between concurrently executing software components and subsystems. As protocols are to support still more complex services that are critical to both the operation of companies and the everyday life of citizens, it is important that they are working correctly already from the initial deployment.

Protocol engineering [80] typically involves a specification of the *service* that the protocol is to provide. Through a synthesis or design step, a protocol design is developed with the aim of providing the desired service. For protocol design, *functional* and *performance* validation can be conducted to investigate and reason about the properties of the design. Functional validation focuses on the

⋆ Work supported by the Research Council of Norway project 194521 (FORMGRID)

logical correctness of the protocol such as the absence of deadlocks and livelocks, that a request is always followed by a response, or whether the proposed protocol design provides the desired services. Performance validation is concerned with quantitative properties such as delays, throughput, and response time. Eventually, the protocol design is implemented and may then be subject to further testing.

Protocol design is in many cases a challenging task. One reason for this is that the execution of a protocol can proceed in different ways, e.g., depending on which messages are lost in transmission, the scheduling of the protocol entities, the time at which events are received from the environment of the protocol, and the execution path taken by the protocol entities. Another reason is that a protocol by nature involves independently scheduled entities which makes testing and reproduction of executions difficult. All this means that protocols often have a very large number of possible executions. In this process, it is easy for a protocol engineer to overlook important interaction patterns which may in turn lead to gaps or malfunction of the protocol.

The specification of the protocol service and the protocol design is, in many cases, based on natural language descriptions. One example of this is the Request for Comments (RFC) documents published by the Internet Engineering Task Force (IETF) [47]. Natural language specifications of protocols often have many issues that needs to be resolved before a properly working implementation can be obtained. One class of issues originates from the fact that such specifications are inherently ambiguous making it difficult to achieve inter-operability between independent implementations. Another source of issues to resolve is that the specifications are often incomplete in that the behaviour of the protocol is not described for all cases.

The challenges outlined above have made protocols a prominent application domain for formal description techniques [46], including Petri Nets [93, 97]. In this paper we concentrate on the use of Coloured Petri Nets (CPNs) [56, 61, 59] for modelling and functional validation of protocol designs. Our purpose is to provide an introduction to, and an overview of, how CPNs have been applied for practical validation of protocol designs. We approach this by presenting selected parts of CPN models and associated results originating from projects conducted in an industrial context with industrial-sized protocols. More specifically, we present in the core of this paper the application of the CPN modelling language, tools, and techniques for functional validation of the following protocols:

The DYMO Routing Protocol. The Dynamic On-Demand Routing Protocol for Mobile Ad-hoc Networks (DYMO) [15] is a routing protocol for mobile ad-hoc networks being developed by the MANET working group of the IETF. The DYMO case study is used to illustrate *protocol modelling* with CPNs and to introduce the basic constructs of the CPN modelling language. Our presentation is based on the CPN model constructed in a project on modelling and validating DYMO [25].

The Generic Access Networks (GAN) Architecture. The GAN protocol architecture [2] is developed by the 3rd Generation Partnership Project (3GPP)

for accessing telephone services via Internet Protocol (IP) networks. The GAN case study is used to introduce the basics of *explicit state space exploration* and show how it can be used in a fully automatic manner as a first step in the *verification* of a protocol design. The presentation is based on the project [30] conducted at TietoEnator A/S to specify the detailed usage of protocol software and services via specialisation of the GAN protocol architecture.

The Routing Interoperability Protocol (RIP). The RIP protocol developed at Ericsson Telebit A/S enables routing of IP packets between core IP networks and mobile ad-hoc networks. The RIP case study is used to illustrate how *application-specific behavioural visualisation* can be applied on top of CPN models. In particular, how it can be used to obtain a first executable prototype of the protocol design allowing for early experiments and for presentation to customers and management with the aim of soliciting protocol design requirements. Our presentation is based on the project [74] conducted in cooperation with Ericsson where CPN modelling was used to specify the operation of the RIP protocol.

The Edge Router Discovery Protocol (ERDP). The ERDP protocol is an IPv6-based protocol allowing edge routers to configure gateways in mobile ad-hoc networks with IP address prefixes. The ERDP case study is used to illustrate how the combined use of CPN modelling, state space exploration, and behavioural visualisation all contributed to identify and resolve design issues and errors during ERDP development. Our presentation is based on the project [67] conducted at Ericsson Telebit A/S on the design of the ERDP protocol.

The rest of this paper is organised as follows. Section 2 provides a high-level overview of CPNs and related techniques used for functional validation of protocol designs. Sections 3-6 then present the application of CPNs on the four protocols introduced above. In Sect. 7 we survey related work where CPNs have been used for protocol validation. Finally, Sect. 8 contains conclusions and outlines directions for future work. The reader is assumed to be familiar with the basic ideas of Petri nets [97] and TCP/IP communication protocols [21]. The reader is referred to [59] for a comprehensive introduction to CPNs, state space exploration, and behavioural visualisation of CPN models.

2 Background: CPNs and Functional Protocol Validation

The CPN modelling language belongs to the family of High-level Petri Nets and combine Petri Nets with the Standard ML (SML) programming language [100]. Petri Nets provide the foundation of the graphical notation and the semantical foundation for modelling concurrency, synchronisation, and communication. The functional programming language SML provides primitives for representing sequential aspects of protocols (such as data manipulation) and for creating compact and parameterisable models. Formal modelling and validation with CPNs

is supported by CPN Tools [95] which provides support for construction, simulation, functional and simulation-based performance analysis of CPN models. The addition of data types and a high-level programming language offered by CPN (in contrast to ordinary Petri nets) is highly important when constructing Petri net models of protocols. As an example, with ordinary Petri nets each message type exchanged between protocol entities need to be present with multiple places, and data manipulation (e.g., comparison of data packet content such as sequence numbers) needs to be modelled relying only on net structure resulting in models that are difficult to comprehend.

The advantage of CPNs (and formal description techniques in general) is that they are based on the construction of executable models that make it possible to observe and experiment with the protocol design prior to implementation and deployment using, e.g., simulation. This typically leads to more complete protocol specifications since the model will not be fully operational until all parts of the protocol have been (at least abstractly) specified. Furthermore, the construction of a formal and executable model helps identify and resolve ambiguities that may be present in a natural language specification. Another advantage is the support for model abstractions that makes it possible to specify the operation of the protocol without being concerned with implementation details such as message layout. A model also makes it possible to explore larger scenarios of a protocol system than what is in many cases practically possible in a laboratory.

2.1 Simulation and Behavioural Visualisation

During a protocol model construction phase it is common to use *interactive simulation* of the CPN protocol model to investigate the operation of the protocol in detail. An interactive simulation is similar to single-step debugging and the execution of the CPN model is viewed directly on its graphical representation and provides a simple way of validating that the model operates as intended. In an interactive simulation, the modeller is in charge and determines the next step by selecting between the enabled events in the current state. Interactive simulation is typically combined with the use of *automatic simulation* which is similar to program execution and the purpose is to execute the CPN model without detailed interaction and inspection. Automatic simulation is typically used for testing purposes, and the modeller typically sets up appropriate breakpoints and stop criteria.

Even though the CPN modelling language supports abstraction and hierarchical modules there can still be a significant amount of detail being presented with this approach, and observing every single step either in an interactive simulation or in a log file based on an automatic simulation is often too detailed a level of observation when investigating the behaviour of a model. Furthermore, even if the CPN model is executable, it still lacks the application- and domain-specific appeal of a conventional software prototype. CPN Tools can use the BRITNeY Suite animation framework [111] to create behavioural visualisation [112] and interaction graphics on top of CPN models. The animation framework is a stand-alone application, and CPN Tools invokes the primitives of the

animation framework using remote procedure calls. The animation framework supports a wide range of diagram types via a plug-in architecture that makes it possible to visualise the execution of protocols using both standard diagrams (e.g., message sequence charts) in addition to tailored, application-specific diagrams. In this way it is possible to investigate the behaviour of a protocol design while overcoming the limitations of interactive and automatic simulations. In this paper we give some examples of both standard and application-specific diagrams. The reader is referred to [111] for a comprehensive introduction to the animation framework.

2.2 State Spaces and Verification

Verification of behavioural properties of protocols with CPNs [66] is supported by explicit *state space exploration* [6]. In its simplest form this approach involves computing a directed graph where the nodes corresponds to the set of reachable states of the CPN model and the arcs represent occurrences of events causing state changes. State spaces can be constructed fully automatically by the state space tool in CPN Tools and guarantees complete coverage of all executions. State space hence provides a highly systematic error-detection technique that make it possible to automatically (i.e., algorithmically) check whether a protocol has a formally stated desired property. In addition, state space methods have the advantage that counter examples (error-traces) can be automatically synthesised if the protocol does not satisfy a given property.

The main disadvantage of state space exploration is the inherent state explosion problem [103], and a multitude of advanced state space methods have been developed aimed at alleviating the inherent state explosion problem. Early work on addressing state explosion in the context of CPNs concentrated on computer tool support for, and initial experiments with, the equivalence [57], symmetry [20, 24, 48, 58], and the stubborn set methods [102]. The symmetry and equivalence methods rely on constructing a condensed state space where each node represents an equivalence class of states and each arc represents an equivalence class of events. The symmetry method has, e.g., been applied on a mutual exclusion protocol [62] and an embedded systems protocol [81]. The equivalence method has only been used on a small stop-and-wait protocol [63] due to the obligation of providing a manual soundness proof for the user-provided equivalence relation. The stubborn set method [101, 103] relies on analysing enabling and disabling dependencies between events and use this to explore only a subset of the events in each state encountered during state space exploration. The rich SML-based inscription language which is fundamental building block of the CPN modelling language, however, poses problems for the analysis of transition dependencies in the context of CPNs [72] – unless relying on an unfolding of the CPN model to the equivalent Place/Transition net. Hence, restrictions on the modelling language are required to apply the stubborn set method without relying on unfolding. Another widely used verification approach in the context of CPNs is based is the methodology of [9]. A central component of this approach is an explicit modelling of both the protocol and its service, and the use

of finite-state automata language comparison as a criteria for checking that the protocol conforms to the specified service. Recent work on addressing the state explosion problem in the context of CPNs has concentrated on making more economical use of memory resources when exploring the state space. Memory is (in many cases) the limiting factor in state space exploration of CPN models due to the large state vectors. This work resulted in the development of the sweep-line method [19, 60] and the comback method [110, 27]. The sweep-line suite of methods [19, 69, 68, 8, 83] is aimed at on-the-fly verification and exploits a notion of progress found in many concurrent systems. Exploiting progress allows for the deletion of states from memory during a progress-first traversal of the state space. This in turn reduces peak memory usage. The sweep-line method has been used [41, 105, 34, 35] for the verification of several industrial-sized protocols specified using the CPN modelling language. The comback method can be viewed as an exploration-order independent storage mechanism based on hash compaction [98, 113]. It allows the usually large state vectors of CPN models to be stored in compact form, and the full state vector of a state is reconstructed when needed for comparison with newly generated states. Unlike the classical hash compaction method, the comback method guarantees full coverage of the state space. The ASAP model checking platform [109] has support for a number of these advanced state space methods – including methods developed outside the context of CPNs.

2.3 Formal Specification Techniques for Protocols

CPNs and Petri Nets represents one approach to the formal specification and verification of protocols. Historically, several non-Petri nets based languages targeting protocol specification have been developed, in particular in relation to telecommunication standardisation efforts [75, 94]. The Language of Temporal Ordering Specification (LOTOS) [50, 1, 14] was developed as part of International Standardisation Organisation (ISO) efforts and linked to the development of the Open Systems Interconnection (OSI) reference model. LOTOS is founded on the Calculus of Communicating (CCS) [86] and add a data type component to CCS based on algebraic specification. The Extended State Transition Language (Estelle) [49] also originated from OSI standardisation efforts and is based on extended finite state machines [13] combined with extensions to the PASCAL programming language. The Specification and Description Language (SDL) [55] has evolved in several generations since 1980 within the International Telecommunication Union - Telecommunication Sector (ITU-T). SDL is based on communicating extended state machines and has in later versions been equipped with a formal semantics [55] making it amendable for formal verification. A Unified Modelling Language (UML) Profile [52] linking SDL and UML also exists. A comparison of these classical specification languages can be found in [5]. Estelle, SDL, and CPNs are all equipped with a language for modelling data manipulation, but have a different theoretical foundations (extended state machines versus Petri Nets). Another difference is that CPNs have very few (but still powerful) modelling constructs in contrast to languages such as Estelle and

SDL which have a large and complex set of language constructs to describe the behaviour of protocol entities and their interaction. From this perspective, CPNs provide a simpler and more lightweight approach to protocol modelling which at the same time less implementation specific than, e.g., typical SDL protocol specifications. In that respect, CPNs are close to languages like LOTOS that focus more on abstract and implementation independent protocol specification. Within ITU-T languages has also been developed related to protocol data representation. The Abstract Syntax Notation One (ASN.1) [53] is a notation of describing data structures carried in messages exchanges between protocol entities. The Encoding Control Notation (ECN) [54] is a language for specifying ASN.1 encoding rules. In terms of specification of data structures, the SML data types for defining colour sets in CPNs provide similar capabilities as ASN.1. The Testing and Test Control Notation 3 (TTCN-3) [26] is a language for writing protocol test specification.

The Process Meta Language (Promela) language [46] providing the modelling foundation of the SPIN tool [45] has been widely used for protocol design and verification. Promela is based on Communication Sequential Processes (CSP) [44] and is in contrast to CPNs, a textual modelling language with a different theoretical foundation. In a UML context, state diagrams (charts) [43] are used for modelling protocol modules (e.g., [84]), and message sequence charts (MSCs) [51] (sequence diagrams in UML) are being used in particular for specifying protocols requirements that can later be used in protocol verification [38, 4]. MSCs have also been used for protocol specification using higher-level control flow constructs. In contrast to MSCs which are action-oriented, then state charts and CPNs are both state and action-oriented modelling formalisms. Timed automata [7] as supported, e.g., by the UppAAl tool has also been used for the specification and verification of protocols (e.g., [96, 29]). The UppAal models consists of a network of network of communicating timed automata, and are specifically suited for modelling and verifying protocol where continuous timing constraints are essential. In comparison, the timed concepts provided by CPNs is a discrete time concept of time. An example on the use of CPNs to model protocols with time constraints can be found in [71].

3 The DYMO Protocol

Modelling a protocol involves developing a representation of the *messages* (or packets) exchanged between the *protocol entities*, the *procedure rules* and *internal state* of the protocol entities guarding the processing of messages, and developing a model of the *environment* in which the protocol is being executed. The environment model typically encompass an abstract representation of the communication medium (or channel) over which the protocol operates. The primary purpose of this section is to illustrate how these protocol elements can be represented in the CPN modelling language using the Dynamic On-demand Routing Protocol (DYMO) [15] for mobile ad-hoc networks as an example. This section additionally shows how to construct compact parameterised CPN mod-

els where the number of protocol entities can easily be configured, and how communication networks with a dynamic topology can be modelled.

3.1 MANETs and Operation of the DYMO Protocol

A *mobile ad-hoc network* (MANET) comprises a collection of mobile nodes, such as laptops, personal digital assistants, and mobile phones, capable of establishing a communication infrastructure for their common use. Ad-hoc networking differs from conventional networks in that the nodes operate in a fully self-configuring, autonomous and distributed manner, without any preexisting communication infrastructure such as base stations and routers. Network layer and routing protocols for ad-hoc networking (including the DYMO protocol) are currently under development by the IETF MANET working group.

The operation of the DYMO protocol consists of two parts: *route discovery* and *route maintenance*. Route discovery is used to establish routes between nodes and begins with an *originator node* multi-casting a Route Request (RREQ) message to all nodes in its immediate range. A RREQ message has a *sequence number* to enable other nodes in the network to judge the freshness of the route request. The ad-hoc network is then flooded with RREQs until the request reaches the *target node* (provided that there exists a path from the originating node to the target node). The target node replies with a Route Reply (RREP) message unicasted hop-by-hop back to the originator node. The route discovery procedure is requested by the Internet Protocol (IP) layer on a node when it receives an IP packet for transmission and does not have a route in its routing table to the target node.

Figure 1(left) depicts the topology of a MANET consisting of six nodes numbered 1–6. An edge between two nodes indicates that the nodes are within direct transmission range. In this case, we assume that all communication links are symmetric. Figure 1(right) (to be discussed below) lists for each node the *routing table* entries created as a result of executing a routing discovery procedure with node 1 as the originator node and node 6 as the target node. The routing table entries in Fig. 1(right) are specified as a pair (*target*, *nexthop*). The second column specifies the entries that are created as a result of a node receiving the RREQ. The third column lists the entries created as a result of receiving the corresponding RREP. When explaining the operation of the DYMO CPN model below, we will use the scenario in Fig. 1 as a running example.

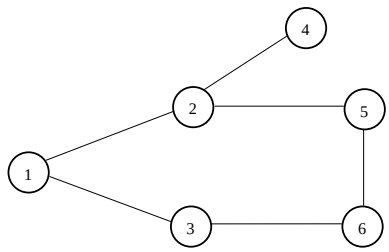
The *message sequence chart* (MSC) in Fig. 2 depicts one possible exchange of messages in the DYMO protocol when the originating node 1 establishes a route to target node 6 in the topology in Fig 1(left). Solid arcs represent multi-cast transmission and dashed arcs represent unicast transmission. In the MSC, node 1 multi-casts a RREQ which is received by nodes 2 and 3. When receiving the RREQ from node 1, nodes 2 and 3 create an entry in their routing table specifying a route back to the originator node 1. Since nodes 2 and 3 are not the target of the RREQ they both multi-cast the received RREQ to their neighbours (nodes 1, 4 and 5, and nodes 1 and 6, respectively). Node 1 discards these messages as it was the originator of the RREQ. When nodes 4 and 5 receive

the RREQ they add an entry to their routing table specifying that the originator node 1 can be reached via node 2. When node 6 receives the RREQ from node 3, it discovers that it is the target node of the RREQ, adds an entry to its routing table specifying that node 1 can be reached via node 3, and unicasts a RREP back to node 3. When node 3 receives the RREP it adds an entry to its routing table stating that node 6 is within direct range, and use its entry in the routing table that was created when the RREQ was received to unicast the RREP to node 1. Upon receiving the RREP from node 3, node 1 adds an entry to its routing table specifying that node 6 can be reached using node 3 as the next hop. The RREQ is also multi-casted by node 4, but when node 2 receives it again, it will be discarded by node 2 because it has already processed the RREQ message once. Node 5 also multi-casts the RREQ, but nodes 2 and 6 also discard the RREQ message as it has already been received once. From Fig. 1(right) it can be seen that upon completion of the route discovery procedure, a bidirectional route has been discovered and established between node 1 and node 6 using node 3 as an intermediate hop.

The topology of a MANET changes over time because of the mobility of the nodes. DYMO nodes therefore perform *route maintenance* where each node monitors the links to the nodes it is directly connected to. The DYMO protocol has a mechanism to notify nodes about routes that become broken due to nodes moving out of range of each other. This is done by sending Route Error (RERR) messages which have the effect of informing nodes using the broken route that a new route discovery is needed in order to reestablish a communication path.

3.2 CPN Model Overview and Message Modelling

The DYMO CPN model is a hierarchical model organised in 14 *modules*. Figure 3 shows the *module hierarchy* of the CPN model. Each node in Fig. 3 corresponds to a *module* with System representing the top-level module of the CPN model. An arc leading from one module to another indicates that the latter is a *submodule* of the former. The model is organised into two main parts. The DYMOProtocol module and its nine submodules model the DYMO protocol entities including the internal state of the protocol entities and the procedure rules for receiving messages, internal processing, and sending of messages. The MobileWirelessNetwork



Node	RREQ	RREP
1		(6,3)
2	(1,1)	
3	(1,1)	(6,6)
4	(1,2)	
5	(1,2)	
6	(1,3)	

Fig. 1. Example MANET topology (left) and routing table entries (right).

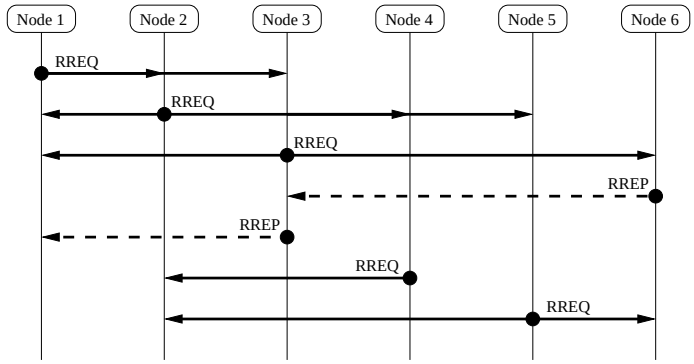


Fig. 2. Message exchange scenario showing DYMO route discovery procedure.

module and its two submodules model the environment for the DYMO protocol. This includes the modelling of how messages are transmitted over a wireless link and the modelling of how the mobility of the nodes affects the current topology of the network. The division of the model into submodules reflects the structure of the DYMO specification [16] and hence maintains a close structural relationship between the natural language specification and the formal CPN model. The CPN model does not capture the transmission of payload from the application layer as the focus of the model is on the route establishment and maintenance of the DYMO protocol.

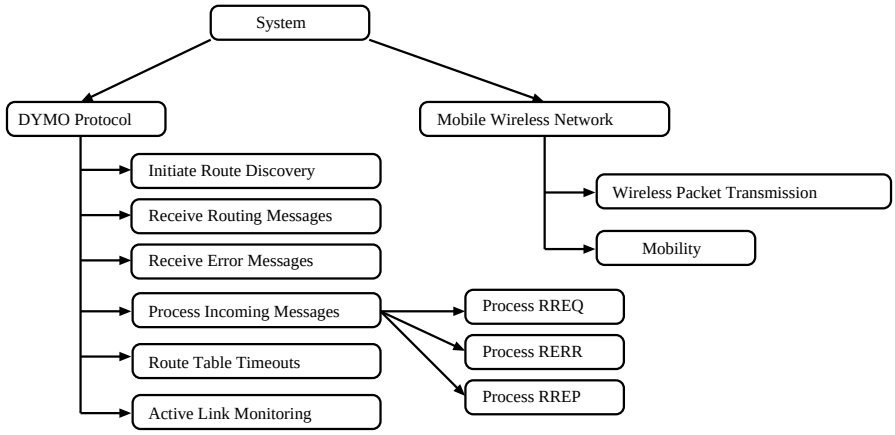


Fig. 3. Module hierarchy for the DYMO CPN model.

The top-level module System is shown in Fig. 4 and is used to connect the two main parts of the model. It corresponds to the System node in Fig. 3. The

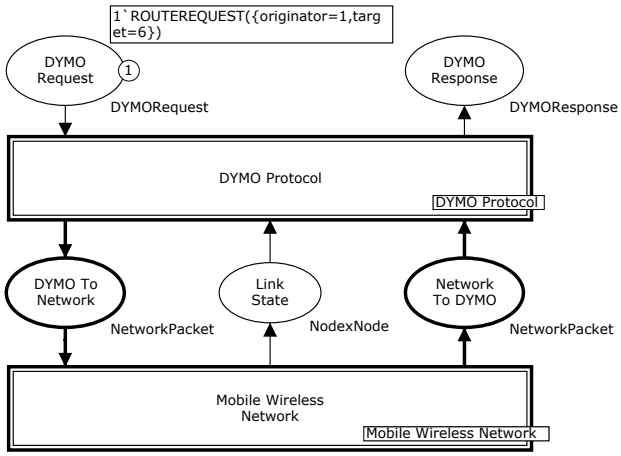


Fig. 4. Top-level System module of the DYMO CPN model.

module has two *substitution transitions* drawn as rectangles with double-line borders. Each of the substitution transitions have an associated tag positioned next to it specifying the name of the associated submodule. The DYMOProtocol substitution transition has the DYMOProtocol module as its associated submodule, and the MobileWirelessNetwork substitution transition has the module MobileWirelessNetwork as its associated submodule. In this model, the substitution transition has the same name as its associated submodule (but this is not generally required).

The two *socket places* DYMOToNetwork and NetworkToDYMO connected to the substitution transition DYMOProtocol are used to model the interaction between the DYMO protocol and the MANET environment as represented by the submodules of the MobileWirelessNetwork substitution transition. The socket place LinkState is used to model the *active link monitoring* that nodes perform to check which neighbour nodes are still reachable. When the DYMO protocol module sends a message, it will appear as a token representing a network packet on the socket place DYMOToNetwork. Similarly, a network packet to be received by the DYMO protocol module will appear as a token on the NetworkToDYMO socket place. Each of the socket places in Fig. 4 (places connected to a substitution transition) is associated with a *port place* in the submodule associated with the substitution transition that the socket place is connected to. The association between a socket and a port place has the effect that the port and the socket places will always have identical markings (tokens). An arc leading to a socket place from a substitution transition means that transitions on the submodule associated with the substitution transitions will add tokens on this place. Analogously, an arc leading from a socket place to a substitution transition means that transitions on the submodule will remove tokens from this place.

The colour set (data types) of each place determining the kind of tokens that can reside on the place is written below each place. The colour set declarations used in Fig. 4 is provided in Fig. 5. A record colour set is used for representing the packets transmitted over the wireless links. A `NetworkPacket` consists of a source (field `src`), a destination (field `dest`), and some `data` (payload). The DYMO messages are designed to be carried in User Datagram Protocol (UDP) datagrams. This means that the network packets are abstract representations of IP/UDP datagrams. The model abstracts from all fields in the IP and UDP datagrams (except source and destination fields) as only these impact the DYMO protocol logic. The source and destination of a network packet are modelled by the `IPAddr` colour set. There are two kinds of IP addresses in the model: `UNICAST` addresses and the `LL_MANET_ROUTERS` multi-cast address. The multi-cast address is used, e.g., in route discovery when a node is sending a RREQ to all its neighbouring nodes. Unicast addresses are used as source of network packets and, e.g., as destinations in RREP messages. A unicast address is represented as an integer from the colour set `Node`. Hence, the model abstracts from real IP addresses and identify nodes (communication interfaces) using integers in the interval $[1; N]$ where `N` is a model parameter specifying the number of nodes in the MANET.

```
(* --- Nodes and abstract IP/UDP messages --- *)
colset Node          = int with 0 .. N;
colset IPAddr        = union UNICAST : Node + LL_MANET_ROUTERS;

colset NetworkPacket = record src  : IPAddr * dest : IPAddr *
                              data : DYMOMessage;

(* --- DYMO service --- *)
colset RouteRequest = record originator : Node * target : Node;

colset DYMORequest  = union ROUTEREQUEST : RouteRequest;

colset RouteResponse = record originator : Node * target : Node *
                              status     : BOOL;

colset DYMOResponse  = union ROUTERESPONSE : RouteResponse;
```

Fig. 5. Colour set declarations for nodes, network packets, and DYMO service.

The two places DYMORequest and DYMOResponse in Fig. 4 are used to interact with the service provided by the DYMO protocol. A route discovery for a specific destination is requested by putting a token on the DYMORequest place and a DYMO response to a route discovery request is then provided by DYMO

as a token via the DYMOResponse place. The colour sets `DYMORequest` (see Fig. 5) specifies the identity of the `originator` node requesting the route and the identity of the `target` node to which a route is to be discovered. Similarly, a `DYMOResponse` message contains a specification of the `originator`, the `target`, and a boolean `status` specifying whether the route discovery was successful. The colour sets `DYMORequest` and `DYMOResponse` are defined as union types to make it easy to later extend the model with additional requests and responses. By setting the *initial marking* of the place DYMORequest, it can be controlled which route discovery requests are to be made.

The small circles and associated boxes in Fig. 4 show the *current marking* of the CPN model. The small circle positioned inside a place indicates the number of tokens on the place in the current marking. In Fig. 4, there is a single token on the place DYMORequest with colour `ROUTEREQUEST({originator=1,target=6})` as specified in the box positioned next to the small circle. This marking corresponds to the DYMO protocol being requested to establish a route from node 1 to node 6 as considered in the scenario in Fig. 1.

3.3 Modelling the DYMO Protocol Entities

The top-level module for the DYMO protocol part of the CPN model is the DYMOProtocol module shown in Fig. 6. The module has five substitution transitions modelling initiating route requests (substitution transition InitiateRouteDiscovery), reception of RREQ and RREP messages (substitution transition ReceiveRoutingMessages), the reception of RERRs (substitution transition ReceiveErrorMessages), processing of incoming messages (substitution transition ProcessIncomingMessages), and timer management associated with the routing table entries (substitution transition RouteTableEntryTimeouts). The places DYMORequest, LinkState, and NetworkToDYMO are *input port places* of the module as indicated by the In tag positioned next to them. Each of these places are associated with the accordingly named socket places in Fig. 4. Similarly, the places DYMOToNetwork and DYMOResponse are *output port places* as indicated by the Out tag positioned next to them, and they are associated to the accordingly named socket places in Fig. 4

All submodules of the substitution transitions in Fig. 6 create and manipulate DYMO messages which are represented by the colour sets defined in Fig. 7. The definition of the colour sets used for modelling the DYMO messages is based on a direct translation of the description of DYMO messages as found in the DYMO specification [16]. In particular, the same names of message fields as in [16] have been used. The model abstracts from the compact packet layout defined for the DYMO protocol. This is done to ease the readability of the CPN model, and since the packet layout is not important when considering only the functional operation of the DYMO protocol.

The place RoutingTable and the place OwnSeqNum are used to model the routing table and the sequence number of nodes, respectively, that are maintained as part of the internal state of each mobile node. In the marking depicted in Fig. 6 both of these places contain a multi-set containing six tokens. Within

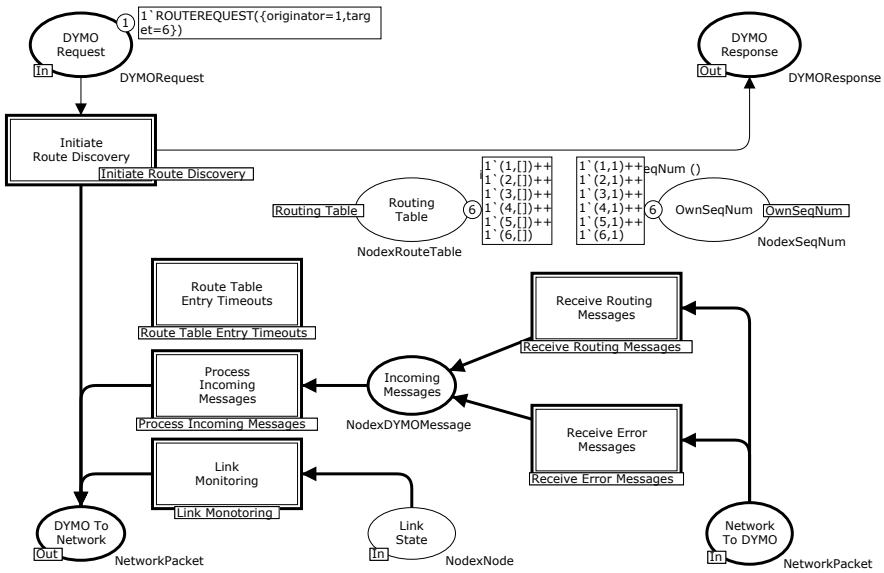


Fig. 6. The DYMOProtocol module.

the boxes specifying the colours of the individual tokens, ++ (pronounced and) is used to denote union of multi-sets and ‘ (pronounced of) is used to specify the coefficients (i.e., the number of occurrences of tokens with a given colour). The colour set `SeqNum` used to represent the sequence number of a node was defined above, and the colour set `RouteTable` is defined in Fig. 8. To allow each node to have its own sequence number, we use the colour set `NodexSeqNum`. The marking in Fig. 6 corresponds to a MANET with six mobile nodes. The first component of each token on the place OwnSeqNum specifies the identity of a node and the second component specifies the sequence number of the node. Initially, the sequence number of all nodes is set to one. Similarly, it can be seen that the routing table of each mobile node is empty as represented by the empty list (`[]`) specified for each node in the marking of RoutingTable.

The submodules of the DYMOProtocol module all need to access the routing table and the sequence number maintained by each node. To reduce the number of arcs in the modules, the routing table and the sequence numbers have been modelled using *fusion sets*. A fusion set allows a set of places in different modules to be linked together into one compound place across the hierarchical structure of the model. In this case, we have a fusion set OwnSeqNum (for linking together places modelling the sequence number of each node) and a fusion set RoutingTable (for linking the places modelling the routing table of each node). The name of the fusion set which a place belongs to (if any) is written in a tag positioned next to the place.

```
colset SeqNum          = int with 0 .. 65535;
colset NodexSeqNum     = product Node * SeqNum;
colset NodexSeqNumList = list NodexSeqNum;

colset RERRMessage = record HopLimit         : INT *
                            UnreachableNodes : NodexSeqNumList;

colset RoutingMessage = record TargetAddr : Node   * OrigAddr   : Nodes *
                               OrigSeqNum : SeqNum * HopLimit   : INT   *
                               Dist       : INT;

colset DYMOMessage = union RREQ : RoutingMessage + RREP : RoutingMessage +
                           RERR : RERRMessage;
```

Fig. 7. Colour set declarations for DYMO messages.

```
colset RouteTableEntry = record
                           Address        : IPAddr * SeqNum : SeqNum *
                           NextHopAddress : IPAddr * Broken : BOOL   *
                           Dist           : INT;

colset RouteTable      = list RouteTableEntry;
colset NodexRouteTable = product Node * RouteTable;
```

Fig. 8. Colour set declarations for routing table entries.

Initiate Route Discovery Module. We consider the InitiateRouteDiscovery module shown in Fig. 9 as a representative example of a submodule at the most detailed level of the CPN model. This module specifies how the route discovery procedure is initiated when a request for a route discovery arrives via the DYMORequest input port. The rectangles in Fig. 9 are ordinary transitions (i.e., non substitution transitions) which means that they can become *enabled* and *occur*. In the marking shown in Fig. 9, a token corresponding to a request for a route discovery originating at node 1 and targeting node 6 is present on the DYMORequest place. In this marking, the transition ProcessRouteRequest is enabled in the following *binding*:

⟨`rreq={originator=1,target=1}`⟩

which binds the *variable* `rreq` of colour set `RouteRequest` to the value in the `ROUTERREQUEST`. Evaluating the input arc expression on the arc from DYMORequest to ProcessRouteRequest results in a multi-set consisting of the single token present on place DYMORequest. The effect of an occurrence of ProcessRequest

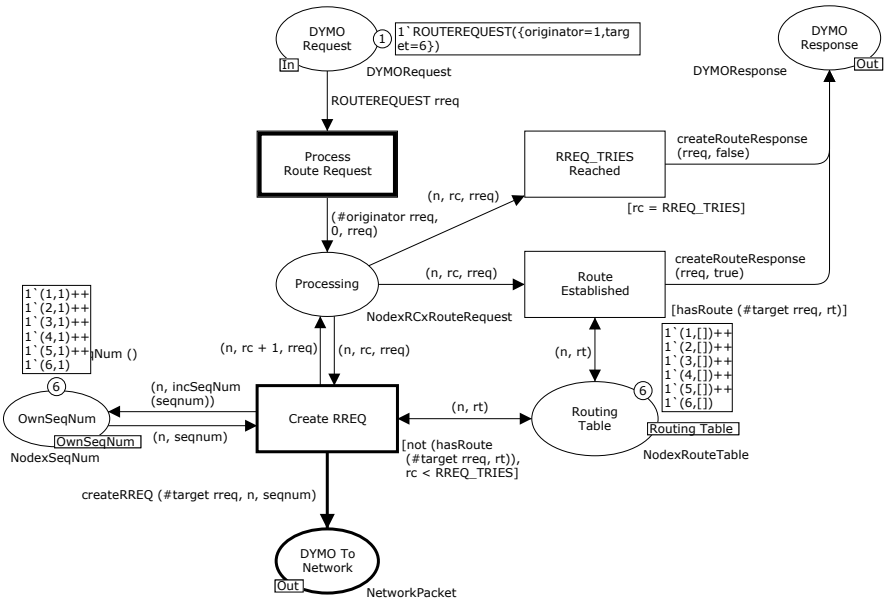


Fig. 9. The Initiate Route Discovery module - initial marking.

with the binding above in the marking in Fig. 9 is that the token on DYMORequest is removed and a token is added to place Processing. The colour of the token added to Processing is obtained by evaluating the *arc expression* on the arc from ProcessRouteRequest to Processing in the binding from above:

```
(#originator rreq,0,rreq)
```

The SML operator `#originator` extracts the originator field in the value bound to `rreq`. The marking resulting from the occurrence of ProcessRouteRequest is shown in Fig. 10. A route request being processed is represented by a token on Processing over the colour set `NodexRCxRouteRequest` which is a product type. The first component of the token on Processing specifies the node processing the route request (i.e., the originator), the second component specifies how many times the RREQ has been retransmitted, and the third component specifies the route request.

In the marking shown shown in Fig. 10, the transition CreateRREQ is enabled with the binding:

⟨`rc=[],rreq={originator=1,target=1},rc=0,n=1,seqnum=1`⟩

The expression in square brackets positioned next to the CreateRREQ transition is a *guard* specifying an additional boolean conditions (beyond the presence of required tokens on input places) for the CreateRREQ transition to be enabled. In this case, the guard specifies that for the transition to be enabled, a route

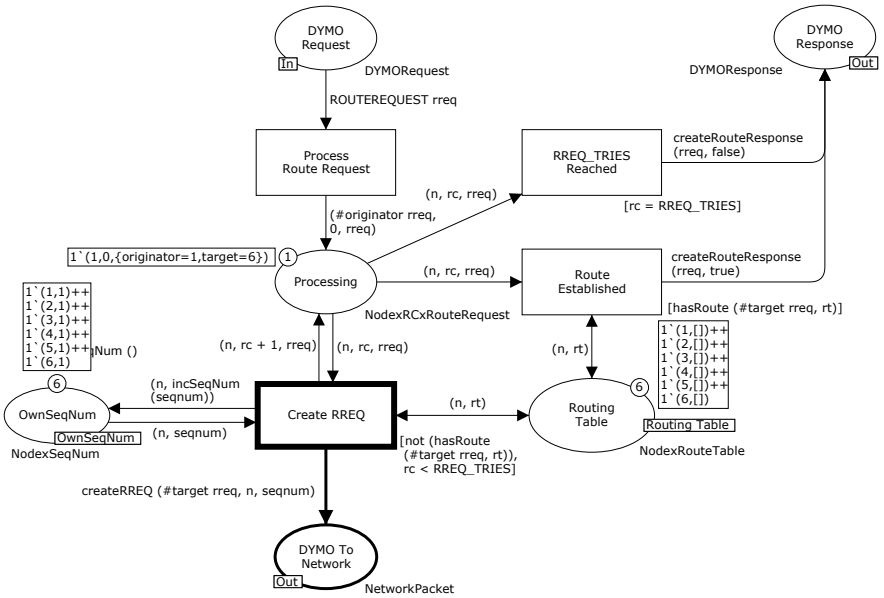


Fig. 10. The Initiate Route Discovery module - after ProcessRouteRequest occurrence.

must not already exist in the route table `rt` to the target node `#target`, and the number of times `rc` the current route request has been retransmitted must be less than the retransmission limit `RREQ_TRIES` for RREQs. The SML function `hasRoute` used in the guard is implemented as follows:

```
fun hasRoute (target, rt:RouteTable) =
    List.exists (fn {Address, ...} => UNICAST(target) = Address) rt
```

and uses the predefined SML function `List.exists` to check whether an entry in the route table `rt` leading to the `target` node already exists. This is a typical example of how SML is used to represent (sequential) data manipulation.

The marking resulting from the occurrence of CreateRREQ is shown in Fig. 11. When sending a RREQ, the sequence number of node 1 sending the request is incremented by 1 and so is the counter specifying how many times the RREQ has been transmitted. Furthermore, a token corresponding to a network packet containing a RREQ message is produced on place DYMOToNetwork. The destination of the packet is set to `LL_MANET_ROUTERS` since it must be sent to all nodes within reach of node 1.

If a route becomes established (i.e., the originator receives a RREP for the RREQ), the RouteEstablished transition becomes enabled and a token can be put on place DYMOResponse indicating that the requested route has been successfully established. If the retransmission limit for RREQs is reached (before a RREP is received), the RREQ_TRIES Reached transition becomes enabled and a

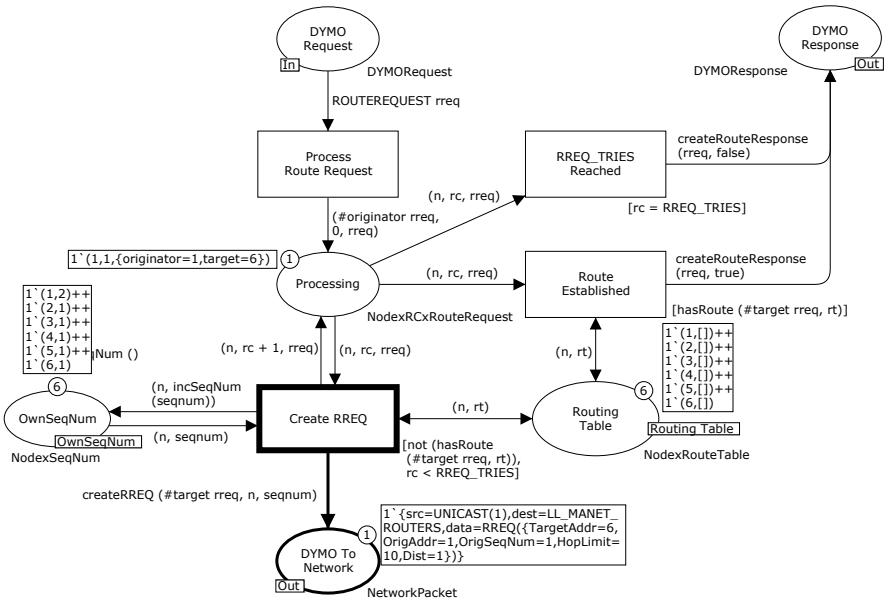


Fig. 11. The Initiate Route Discovery module - after CreateRREQ occurrence.

token can be put on place DYMOResponse indicating that the requested route could not be established.

3.4 Modelling the DYMO Protocol Environment

The MobileWirelessNetwork module shown in Fig. 12 captures the mobile wireless network that DYMO is designed to operate over. It consists of two parts: one part modelling the transmission of network packets represented by the substitution transition WirelessPacketTransmission, and one part representing the mobility of the nodes represented by the Mobility substitution transition. The places DYMOToNetwork, NetworkToDYMO, and LinkState are associated to the similarly named socket places in Fig. 4. The transmission of network packets is done relative to the current topology of the MANET which is explicitly represented via the current marking of the Topology place. The topology is represented using the colour set `Topology` defined in Fig. 13.

The idea is that each node has an adjacency list of nodes that it can reach in one hop, i.e., its neighbouring nodes. The marking of place Topology in Fig. 12 corresponds to the topology in Fig. 1(left). This adjacency list is then consulted when a network packet is being transmitted from a node to determine the set of nodes that can receive the network packet. In this way, the dynamic topology is modelled by the addition and removal of nodes from the adjacency lists. The

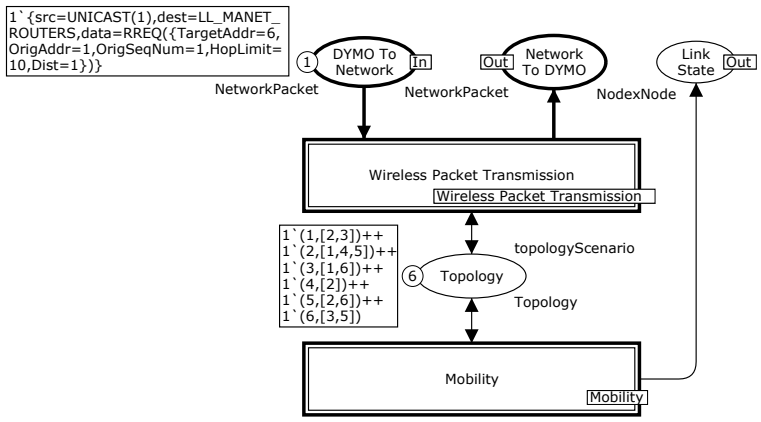


Fig. 12. The Mobile Wireless Network.

place LinkState models that a node can be informed about the reachability of its neighbouring nodes which is used in active link monitoring.

```
colset NodeList = list Node;
colset Topology = product Node * NodeList;
```

Fig. 13. Colour set declarations for topology modelling.

The WirelessPacketTransmission module models the actual transmission of packets and is shown in Fig. 14. The module captures how network packets are transmitted via the physical network from one node to the next. Packets are transmitted over the network according to the function transmit on the arc from the transition Transmit to the place NetworkToDYMO. When the Transmit transition occurs in a binding where the boolean variable `success` is set to `true`, then all nodes within reach of the sending node will receive the packet. Otherwise, no nodes will receive the packet. The transition Transmit is enabled in the marking shown in Fig. 14 (left) and the marking resulting from a successful transmission of the packet on DYMOToNetwork is shown in Fig. 14 (right). In this case two tokens are added to place NetworkToDYMO corresponding to nodes 2 and 3 receiving the packet being multi-casted from node 1.

In a real network, a transmission could be received by any subset of the neighbouring nodes (e.g., because of signal interference). Here it is only modelled that either all of the neighbouring nodes receive the packet or none of the nodes receive it. This is sufficient because the modelling of the dynamic topology means

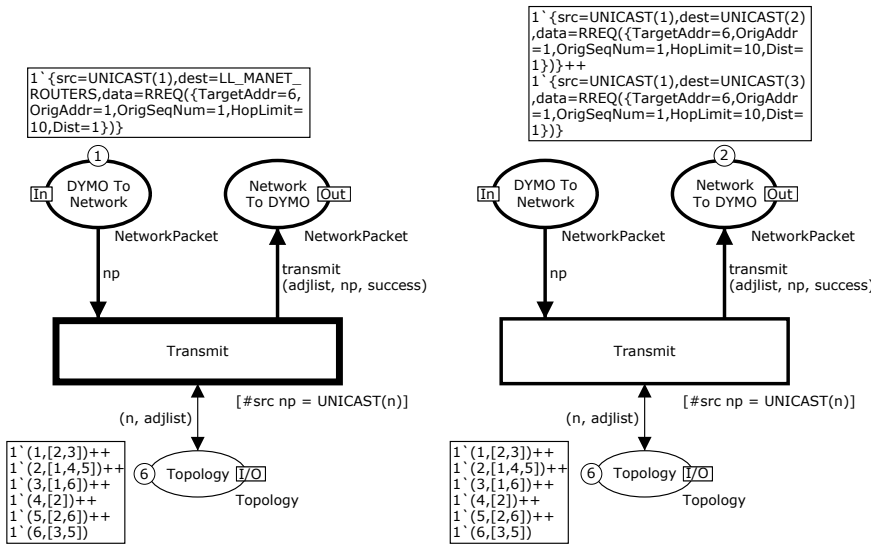


Fig. 14. Transmission of packets - before (left) and after (right) transmission.

that a node can move out of reach of the transmitting node immediately before the transmission occurs which has exactly the same effect as a signal interference in that the node does not receive the packet. Hence, signal interference and similar phenomena imply that a node does not receive a packet is in the model equivalent to the node moving out of reach of the transmitting node.

3.5 Lessons Learned and Perspectives

The development of the DYMO CPN model was based on the natural language specification provided in the Internet draft [15] specifying the DYMO protocol. The modelling work was done when version 10 [15] was the most recent DYMO specification. In the process of constructing the CPN model and simulating it, several issues and ambiguities in the specification were discovered. The most important ones are summarised in Table 1. These issues were submitted to the IETF MANET Working Group mailing list [82] and issue 1 and 3-7 were acknowledged by the DYMO developers and taken into account in the subsequent version DYMO specification [16] (version 11). Issue 2 was not considered critical as it causes route discovery to fail in scenarios which according to the experience of the DYMO developers would seldom occur in practise.

The modelling conducted with the DYMO protocol illustrates that the construction of a formal and executable model provides a very systematic and comprehensive way of reviewing a protocol design document (such as the DYMO Internet draft) and how it can contribute to increasing the quality of a protocol design specification. Similar conclusions can also be drawn from other case

Table 1. DYMO CPN modelling [25]: issues identified in the modelling phase.

Issue	Description
1	When processing a routing message, a DYMO router may respond with a RREQ flood, i.e., a RREQ addressed to the node itself, when it is target for a RREQ message (cf. [15], Sect. 5.3.4). It was not clear from the specification which information to put in the RREQ message, i.e., the originator address, hop limit, and sequence number of the RREQ.
2	When judging the usefulness of routing information, the target node is not considered. This means that a new request with a higher sequence number can make an older request for another node stale since the sequence number in the old message is smaller than the sequence number found in the routing table.
3	When creating a RREQ message the distance field in the message is set to zero. This means that for a given node n an entry in the routing table of a node n' connected directly to n may have a distance to n which is 0. Distance is a metric indicating the distance traversed before reaching n, and the distance between two directly connected nodes should be one.
4	In the description of the data structure route table entry (cf. [15], Sect. 4.1) it is suggested that the address field can contain more than one node. It was not clear why this was the case.
5	When processing RERR messages (cf. [15], Sect. 5.5.4) it is not specified whether the hop limit shall be decremented.
6	When retransmitting a RREQ message (cf. [15], Sect. 5.4), it was not explicitly stated whether the node sequence number should be increased.
7	Version 10 of DYMO introduced the concept of distance instead of hop count. Distance is a more general metric, but in the routing message processing (cf. [16], Sect. 5.3.4) it is incremented by one. We believe it should be up to the implementers how much distance is incremented depending on the metric used.

studies where CPN modelling has been applied to protocols developed in the context of IETF. A CPN model of the DYMO protocol has also been developed in [12] where a considerably more compact CPN model of the DYMO protocol directly targeting state space exploration was developed. A number of other issues related to the functionality of the DYMO protocol were reported in [12]. In comparison to the CPN model in this section, the CPN model developed in [12] provides a more abstract modelling approach that does not use an explicit representation of MANET topology.

4 The GAN Protocol Architecture

This section focuses on how standard behavioural properties of CPNs in combination with explicit state space exploration can be used to verify basic properties of protocols. Furthermore, this section gives an example of how CPNs can be used to model a system spanning multiple protocols and protocol layers. The presentation is based on a project [30] in which CPN modelling and state space exploration was used at TietoEnator Denmark in early phases of developing an implementation corresponding to a particular instantiation [42] of the generic GAN architecture [2] aimed at integrating IP and telephone services.

4.1 GAN Secure Connection Establishment

The Generic Access Network (GAN) [2] architecture specified by the 3rd Generation Partnership Project (3GPP) [3] allows access to common telephone services such as SMS and voice-calls via IP networks. A central part of the GAN architecture is the establishment of a secure connection between a *mobile station* (e.g., a mobile phone) and a *GAN controller* through a *security gateway*. The GAN architecture relies on standardised protocols such as Dynamic Host Configuration Protocol (DHCP) for IP address configuration, IP Security (IPsec) [65] for encryption and authentication, and Internet Key Exchange v2 (IKEv2) protocol [64] for negotiation of IPsec parameters.

The purpose of the CPN model constructed in the project was two-fold. Firstly, to define the scope of the protocol software to be developed by TietoEnator. More specifically, the aim was to determine which parts of the generic GAN specification were to be included in the implementation to be developed by TietoEnator. Secondly, to specify the detailed design and usage of the involved protocol software components. The focus of the CPN model is on the establishment of a secure tunnel and the initial GAN message exchanges since this is where important details were not provided in the full GAN specification. In particular, the full GAN specification [2] contained no clear specification of the IKEv2 message exchange and the states that the protocol entities should be in when establishing a GAN connection (at the time of the project in 2007). Furthermore, the GAN specification only states that IKEv2 and IPSec are to be used, and in which operating modes.

4.2 CPN Model of the GAN Protocol Architecture

The CPN model of the secure connection establishment consists of 31 modules organised into four hierarchical levels. In the following, we present four selected modules from the CPN model. Our purpose is to illustrate how the phases that the protocol entities enter when establishing a GAN connection have been modelled, and provide sufficient detail on the CPN model in order for the reader to interpret the verification results presented later. A more in-depth presentation of the CPN model can be found in [30].

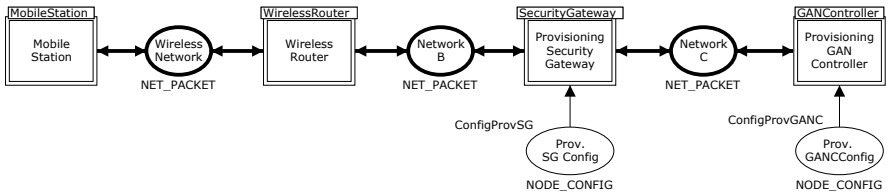


Fig. 15. Top-level module of the GAN model.

Figure 15 shows the top-level module which is organised so that it mimics the GAN network architecture. The substitution transition MobileStation represents the mobile station which is connecting to the telephone network via an IP network. The place Wireless Network connected to MobileStation represents the wireless network which connects the mobile station to a wireless router represented by the substitution transition WirelessRouter. The wireless router is an arbitrary access point with routing functionality, and is connected to the Provisioning Security Gateway, through NetworkB. As part of establishing a GAN connection, an encrypted tunnel is established between the mobile station and the security gateway. The encrypted tunnel is provided by the Encapsulating Security Payload (ESP) mode of the IP security layer (IPSec) [65]. To provide such an encrypted tunnel, both ends have to authenticate each other, and agree on both an encryption algorithm and keys. This is achieved using the Internet Key Exchange v2 (IKEv2) protocol [64]. The provisioning security gateway is connected to the Provisioning GAN Controller via NetworkC. The GAN controllers are connected to the telephone network and perform the relay of traffic to/from the IP networks (NetworkC and the WirelessNetwork). This in turn allows mobile stations to access the services on the telephone network. The places with thin lines connected to the substitution transitions Provisioning Security Gateway and Provisioning GAN Controller are used to provide configuration information to the corresponding network nodes. The CPN model does not include modelling of the telephone network as the scope of the CPN model covers the components involved in establishing the connection with the GAN controller. Furthermore, as the purpose of the model was to represent the protocol entities present on each of the nodes in the network architecture, it sufficed that the model encompassed one mobile node, one wireless router, one provisioning security gateway, and one provisioning GAN controller.

The basic exchange of messages in establishing a GAN connection to the provisioning GAN controller involves three steps. The first step is for the mobile station to acquire an IP address on the wireless network using DHCP. The second phase is to create a secure tunnel to the provisioning security gateway. Having established the secure tunnel, the third phase is for the mobile station to open a secure connection to the GAN controller and register itself. Figure 16 (left) shows the IKEInitiator module of the mobile station and Fig. 16 (right) shows the IKEResponder module of the security gateway. These two peer modules

model the second step of the GAN connection establishment concerned with creating the secure tunnel. Incoming IP packets for the module arrive via the ReceiveBuffer input port places. Outgoing IP packets are put in the SendBuffer places. The states (phases) that the protocol entities goes through during the IKE message exchange when establishing the secure tunnel are represented by the places connecting the substitution transitions.

The state changes are represented by substitution transitions. The submodules of the substitution transitions specify the processing rules for messages during the individual phases. Figure 17 shows the Send IKE_SA_INIT Packet and Handle_SA_INIT_Request modules which are the submodules of the two top-most substitution transitions in Fig 16. The Send IKE_SA_INIT Packet transition in Fig. 17 (left) takes the IKE Initiator from the state Ready to Await IKE_SA_INIT and sends an IKE message to the security gateway initialising the communication. The IP address of the security gateway is retrieved from the Ready place. Figure 17 (right) shows how the IKE_SA_INIT packet is handled by the IKE Responder. The guard of the HandleSA_INIT_Request transition ensures that the transition is only enabled if the incoming packet (token) on IncomingIKERequest represents a IKE_SA_INIT packet. In that case, it sends an IKE packet back to the initiator as specified by the arc expression on the arc from Handle_SA_INITRequest and the responder enters the Wait for EAP Auth state. The submodules of the other substitution transitions in Fig. 16 are similar.

The establishment of a GAN connection involves multiple layers of the IP network stack. DHCP (used to configure the mobile station) and GAN are application layer protocols, IKE is a transport layer protocol, and IPSec belongs to the network layer. As a consequence, the CPN model of GAN connection spans multiple protocol layers. Furthermore, the protocol entities also access and manipulate the *routing table* and a *security policy database* (SPD) which is maintained at the IP network layer. The establishment of a GAN connection accesses the routing table of a node in order to ensures that packets are put into the secure tunnel, and extracted again at the other end. The SPD describes what packets are allowed to be sent and received by the IP protocol stack, and is also responsible for identifying which packets are to be tunnelled at the mobile station and the security gateway. Each entry in the SPD contains the source and destination addresses to use for matching packets, and an action to perform. Modelled actions are *bypass* (which means allow packet to pass without tunnelling) and *tunnel* (the matched packet is to be sent through an ESP tunnel). As we will see later, the content of the routing table and the SPD play an important role in validating the correctness of the GAN connection establishment. It was therefore required to explicitly represent them in the CPN model.

4.3 Verification of the GAN CPN Model

The goal of applying state space exploration was to verify the completeness of the design. This included verifying that all phases, steps, and messages involved in establishing a secure GAN connection were covered by the design, and the correctness of the connection establishment,i.e., that a GAN connection is

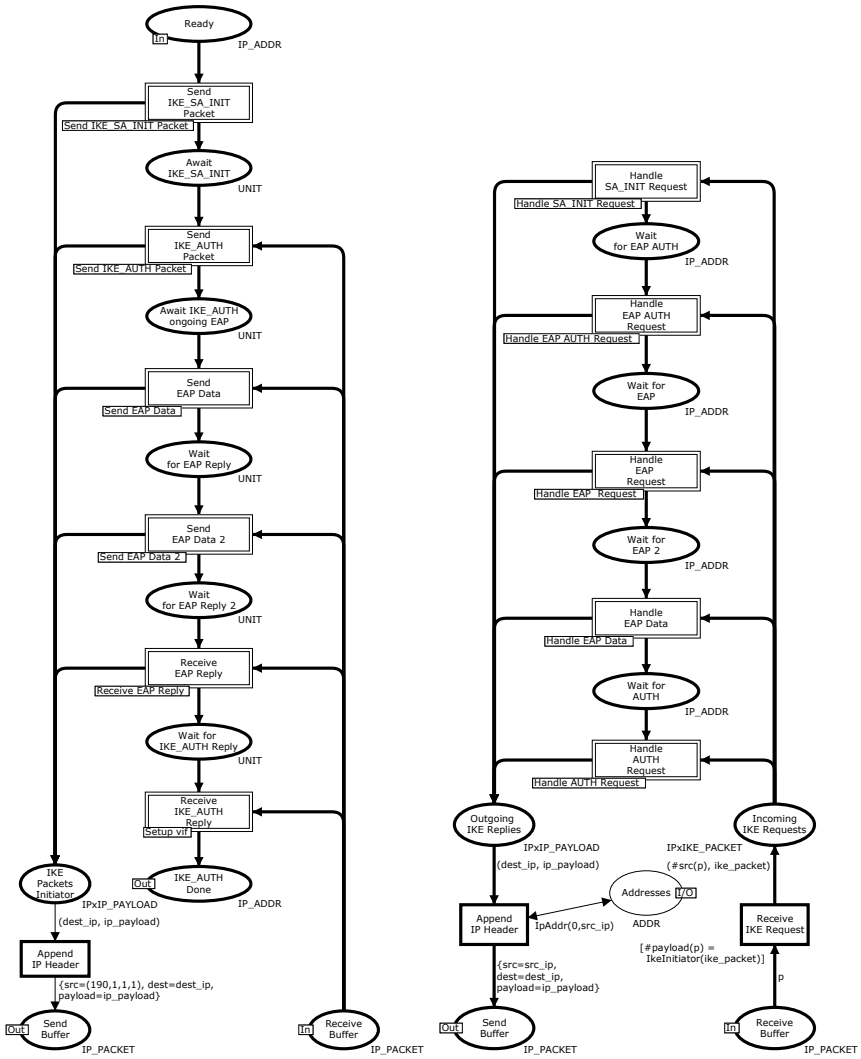


Fig. 16. IKE initiator (left) and IKE responder (right) modules.

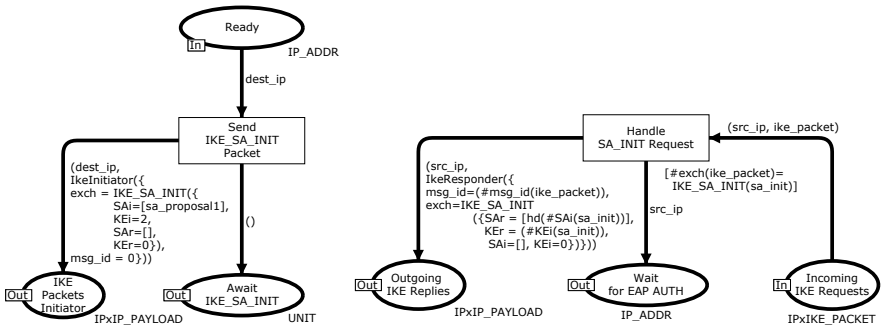


Fig. 17. Example of IKE initiator (left) and IKE responder (right) submodules.

eventually established with the mobile station and the GAN controller being properly synchronised. Verification of the key properties of the design for secure connection establishment was done by *state space exploration*. The basic idea underlying state space exploration is to compute all reachable states and state changes of the CPN model and represent these as a directed graph, where nodes represent markings and arcs represent occurring binding elements. State spaces can be constructed fully automatically by the state space tool in CPN Tools. Verification of the GAN scenario modelling by means of state spaces relied on the use of the *state space report* that can be generated by CPN Tools. The generation of a state space report for the smallest possible configuration of a considered protocol is typically the first step performed when conducting verification of a CPN model.

The state space report is divided into several sections. In the following we present excepts from the individual sections and explain how they can be used for the verification. Figure 18 shows the first part of the state space report for the CPN model. This part provides some *state space statistics* specifying how large the state space is. It can be seen that the state space consists of $3,854$ nodes and $9,225$ arcs. The construction of the state space took 4 seconds. Statistics for the strongly connected component graph (SCC-graph) are also specified. It has $3,514$ nodes and $8,881$ arcs, and was calculated in 2 seconds. The fact that there are fewer nodes in the SCC-graph than in the state space implies that there are non-trivial strongly connected components (SCCs), i.e., SCCs consisting of more than a single state space node. This means that infinite executions exist and that the GAN connection establishment may not terminate. We will investigate the reasons for this at the end of this subsection.

The *boundedness properties* section of the state space report specifies how many and which tokens a place may hold – when considering all reachable states (markings). Figure 19 lists the best upper and best lower integer bounds for selected places in the mobile station module. It can be seen that the first four places modelling the states of the mobile station contain at most one token and may contain zero tokens. Similarly, it can be seen that there is at most one token

in the send, received, and network buffers. The place RoutingTable has a lower integer bound of 0 and an upper integer bound of 1. The lower integer bound is 0 since in the initial marking there are no tokens on this place. During the start-up procedure of the mobile station, a token representing a list of routing table entries is put on this place. The place SecurityPolicyDatabase has a best upper and a best lower integer bound of 1. This means that there is always exactly one token present on this place. This is because the security policy database is modelled as a single token being a list containing the current entries in the security policy database.

The *best upper multi-set bound* of a place specifies for each colour in the colour set of the place the maximal number of tokens that is present on this place with the given colour in any reachable marking. This is specified as a multi-set, where the coefficient of each value is the maximal number of tokens with the given value. If the coefficient is zero, then the colour is omitted in the specification. Figure 20 shows part of the state space report providing the upper multi-set bounds for the security policy databases of the mobile station, wireless router, security gateway, and the GAN controller. The upper multi-set bounds specify the possible tokens that can reside on these places and by carefully inspecting these bounds it was possible to validate that the possible entries in the security policy database were all as desired. Altogether, an inspection of the boundedness properties helped significantly in increasing confidence in the correctness of the design in terms of proper settings of the routing table and the security policy database.

Figure 21 shows the part of the state space report specifying the *home and liveness properties*. The home properties show that there exists a single *home marking*, which is state number 3854. A home marking is a state which can be reached from any reachable state. For the GAN scenario model this means that it is impossible to have an execution sequence starting from the initial state (initial marking) which cannot be extended to reach state 3854. The liveness properties tell us that there is a single *dead marking* which is also state number 3854. A dead marking is a state in which no transitions are enabled. This means that the marking corresponding to node 3854 is both a home and a dead marking.

To obtain information about the marking corresponding to node number 3854, the node number was transferred into the simulator of CPN Tools and displayed graphically on the CPN model. It was then checked (by inspecting

```
State Space                    Scc Graph
  Nodes:   3,854                 Nodes:   3,514
  Arcs:    9,225                 Arcs:    8,881
  Secs:        4                 Secs:        2
```

Fig. 18. State space report – statistics.

```
Best Integer Bounds         Upper      Lower
   Down                      1          0
   Ready                     1          0
   VIF open to Prov. SG      1          0
   VIF Closed                1          0

   Send Buffer               1          0
   Receive Buffer            1          0
   Network Buffer            1          0

   Routing Table             1          0
   Security Policy Database  1          1
```

Fig. 19. State space report – integer bounds.

the markings of the individual places) that the marking corresponded to the desired terminating state of the GAN connection establishment procedure, i.e., the state where the mobile station has obtained an IP address, has successfully communicated with the provisioning GAN controller, all protocol modules are in a state corresponding to the GAN connection having been established, and the routing tables and security databases contain the correct entries. The fact that state 3854 is the only dead marking tells us that the protocol as specified by the CPN model is partially correct, i.e., if execution terminates we have the correct result. Furthermore, because node 3854 is also a home marking it is always possible to terminate the GAN connection establishment with the correct result.

The analysis above showed that it is always possible to terminate the GAN connection establishment procedure correctly, but there is no guarantee that it will eventually happen. The section of the state space report providing information about *fairness properties* showed that the two transitions RejectDiscoveryRequest and HandleGARCReject which are part of the GAN controller module were *impartial*. This means that these two transitions occur infinitely often in any infinite occurrence sequence. The two transition occurs if the GAN controller decides to reject an incoming connection from a mobile station. Hence, if the connection establishment procedure does not terminate in the single home and dead marking identified, then it is because the GAN controller keeps rejecting the connection.

4.4 Lessons Learned and Perspectives

The validation of secure connection establishment in the considered GAN scenario is representative for how validation of protocols is typically performed with CPN Tools – as it in practise involves a combination of both simulation and state space exploration. As part of the construction of the GAN model, the support for interactive simulation in CPN Tools was used to perform detailed

```
Mobile Station: Security Policy Database
    1‘[{src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=PayloadList([PAYLOAD_DHCP]),policy=SpdBypass}]++
    1‘[{src=((80,1,1,1),32),dest=((12,0,0,0),8),
        nl_info=AnyNextLayer,policy=ESPTunnel(((190,1,1,1),(172,1,1,2)))},
       {src=((190,1,1,1),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]++
    1‘[{src=((190,1,1,1),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]

Wireless Router: Security Policy_Database
    1‘[{src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]

Security Gateway: Security Policy Database
    1‘[{src=((13,0,0,0),8),dest=((80,1,1,1),32),
        nl_info=AnyNextLayer,policy=ESPTunnel(((172,1,1,2),(190,1,1,1)))},
       {src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]

GAN Controller: Security Policy Database
    1‘[{src=((0,0,0,0),0),dest=((0,0,0,0),0),
        nl_info=AnyNextLayer,policy=SpdBypass}]
```

Fig. 20. State space report – best upper multi-set bounds.

checks to ensure that the model behaviour was as desired. Even though the use of interactive simulations (and simulation in general) cannot be used to prove correct behaviour, it proved to be very useful in identifying situations related to improper manipulations of the entries in the routing tables and security policy database - or when additional detail not present in the GAN specification had to be worked out and specified. Furthermore, interactive simulation was helpful in identifying issues that led the GAN connection establishment procedure to terminate prematurely, e.g., because a certain phase of the connection establishment was missing in the CPN model. These issues manifested themselves in markings where the GAN connection had not yet been established, but where

```
Home Properties                 Liveness Properties
  Home Markings: [3854]           Dead Markings: [3854]
```

Fig. 21. State space report – home and liveness properties.

no transitions were enabled. This was in particular effective in making explicit where further specification of the message exchanges were required.

The interactive simulation was in later phases replaced with automatic simulation where a number of random executions of the CPN model were performed with the purpose of checking whether the execution of the CPN model resulted in a state in which the GAN connection was properly established. Eventually state space exploration of the CPN model was conducted which succeeded in establishing the key property that a GAN connection will eventually be established provided that the GAN controller does not keep rejecting the connection request. The verification conducted also illustrated the general observation that in many cases, the use of basic state space exploration and the state space report (i.e., investigating standard behavioural properties of Petri nets) are sufficient in establishing key properties of a protocol design. In this case the state space was small in size and could be generated in a few seconds without the use of advanced state space exploration techniques.

5 The Routing Interoperability Protocol

The section show how a CPN model can be augmented with application-specific behavioural visualisation reflecting the execution of the CPN model. This section is based on a project conducted at Ericsson Telebit A/S addressing the specification of the Routing Interoperability Protocol (RIP)[3] for routing packets between IP core networks and mobile ad-hoc networks. The CPN model of RIP augmented with behavioural visualisation was used as an early model-based prototype of RIP. It allowed the protocol design to be discussed among protocol engineers unfamiliar with CPNs, and it also enabled the protocol design to be presented to customers with the purpose of soliciting requirements of the services to be provided by the protocol.

5.1 CPN Model of the RIP Protocol

The main purpose of the routing interoperability protocol is to ensure that a packet flow between a host in the core network and a mobile node in an ad-hoc network is always relayed via one of the closest gateways that connect the core network and the mobile ad-hoc network. Figure 22 shows the top level module of the CPN model which reflects the network architecture that the RIP protocol is designed to operate in. The network architecture consists of three parts: an IPv6 core network represented by the CoreNetwork substitution transition (left) and its submodules, a mobile ad-hoc network represented by the AdHocNetwork substitution transition (right) and its submodules, and two gateways represented by the substitution transitions Gateway1 and Gateway2. The basic idea in the interoperability protocol is that the mobile nodes register the IPv6 address in

[3] RIP as discussed in this section should not be confused with the Routing Information Protocol[85]

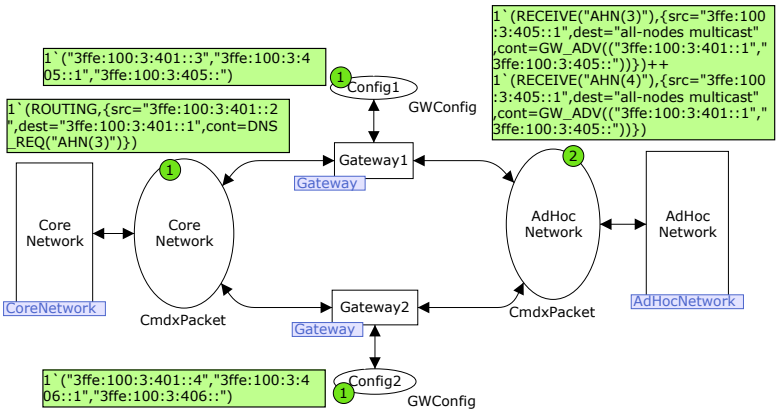


Fig. 22. The System module – top-level module of the CPN model.

the Domain Name Server (DNS) server in the core network that corresponds to an IPv6 address prefix announced by the closest (preferred) gateway. Updates to the DNS database managed by the DNS server rely on the Dynamic Domain Name System Protocol [108].

The places CoreNetwork and AdHocNetwork are used for modelling the packets in transit on the core network and ad-hoc network, respectively. Figure 22 depicts a state in which there is one token on place CoreNetwork and two tokens on place AdHocNetwork. As an example, place CoreNetwork contains one token with the colour:

```
(RECEIVE("3ffe:100:3:401::1"), {src="3ffe:100:3:401::2",
       dest="3ffe:100:3:401::1",cont=DNSREQ("AHN(3)")})
```

representing a DNS request (`DNSREQ`) in transit on the core network from a host with source IPv6 address `3ffe:100:3:401::2` to a DNS server with destination IPv6 address `3ffe:100:3:401::1`. IPv6 addresses are 128-bit and by convention written in hexadecimal notation in groups of 16-bits separated by a colon (:). Leading zeros are skipped within each group and a double colon (::) is a shorthand for a sequence of zeros. Addresses consist of an *address prefix* and an *interface identifier*.

The place AdHocNetwork contains two tokens representing gateway advertisements in transit to nodes in the ad-hoc network. The gateways periodically announce their presence to nodes in the mobile ad-hoc network by sending *gateway advertisements* containing an IPv6 *address prefix*. The two Config places contain a token representing the configuration of the corresponding gateway. It consists of the IPv6 address of the gateway interface connected to the core network, the IPv6 address of the gateway interface connected to the ad-hoc network, and the address prefix announced by the gateway. Address prefixes are written

in the form x/y where x is an IPv6 address and y is the length of the prefix. The mobile nodes in the ad-hoc network configure IPv6 addresses based on the received gateway advertisements. In the marking depicted in Fig. 22, Gateway1 is announcing the 64-bit address prefix 3ffe:100:3:405::/64 and Gateway2 is announcing the prefix 3ffe:100:4:406::/64. Each of the gateways has configured an address on the interface to the ad-hoc network based on the prefix they are announcing to the ad-hoc network. Gateway1 has configured the address 3ffe:100:3:405::1 and Gateway2 has configured the address 3ffe:100:3:406::1. The gateways have also configured addresses on the interface to the core network based on the 3ffe:100:3:401::/64 prefix of the core network.

Figure 23 lists the definitions of the colour sets used in the System module. IP addresses, prefixes, and symbolic IP addresses are represented by colour sets IPAdr, Prefix, and Symname all defined as the set of strings. The colour set PacketCont and Packet are used for modelling the IP packets. The five different kinds of packets used in RIP are modelled by the PacketCont colour set:

`DNS_REQ` modelling a DNS request packet. It contains the symbolic IP address to be resolved to a (numerical) IP address by a DNS server.

`DNS_REP` modelling a DNS reply. It contains the symbolic IP address and the resolved IP address.

`DNS_UPD` modelling a DNS update. It contains the symbolic IP address to be updated and the new IP address to be bound to the symbolic address.

`GW_ADV` modelling the advertisements disseminated from the gateways. An advertisement contains the IP address of the gateway and the announced prefix.

`PACKET` modelling generic payload packets belonging to packet flows between hosts and the mobile nodes.

The colour set `Packet` models the packets as a record containing the source, destination, and content. The actual payload (content) and layout of packets are not essential for modelling the interoperability protocol and has therefore been abstracted away. The colour set Cmd is used to control the operation of the various modules in the CPN model. The colour set GWConfig models the configuration information of the gateway.

The Core Network. Figure 24 shows the CoreNetwork module modelling the core network. This module is the immediate submodule of the substitution transition CoreNetwork of the System module shown in Fig. 22. The port place CoreNetwork is assigned to the CoreNetwork socket place in the System module (see Fig. 22). The substitution transition Routing represents the routing mechanism in the core network. The substitution transition Host represents the host on the core network, and the substitution transition DNS Server represents the DNS server that maintains the DNS database.

The Mobile Ad-hoc Network. Figure 25 depicts the AdHocNetwork module modelling the mobile ad-hoc network. The place Nodes is used to represent the

```
colset Prefix = string;  (* address prefixes *)
colset IPAdr = string;   (* IP addresses     *)
colset SymName = string; (* symbolic names   *)

colset SymNamexIPAdr = product SymName * IPAdr;
colset IPAdrxPrefix = product IPAdr * Prefix;

colset PacketCont = union DNS_REQ : SymName +       (* DNS Request     *)
                          DNS_REP : SymNamexIPAdr + (* DNS Reply       *)
                          DNS_UPD : SymNamexIPAdr + (* DNS Update      *)
                          GW_ADV  : IPAdrxPrefix +  (* Advertisments   *)
                          PACKET;                   (* Generic payload *)

colset Packet = record src  : IPAdr  * dest : IPAdr * cont : PacketCont;
colset Cmd    = union ROUTING              + RECEIVE      : IPAdr +
                      FLOODING     : IPAdr + GWAHNROUTING : IPAdr +
                      AHNGWROUTING : IPAdr;

colset CmdxPacket = product Cmd * Packet;
colset GWConfig   = product IPAdr * IPAdr * Prefix;
```

Fig. 23. Colour set definitions used in the System module.

nodes in the mobile ad-hoc network. The place RoutingInformation is used to represent the routing information in the ad-hoc network which is assumed to be available via some routing protocol executed in the ad-hoc network. This routing information enables among other things the nodes to determine the distance to the reachable gateways. Detailed information about the colour of the token on place RoutingInformation has been omitted.

Figure 26 lists the definition of the colour sets used in the AdHocNetwork module. The colour set AHNConfig is used to model the configuration information for the mobile ad-hoc nodes. Each ad-hoc node is represented by a token on place

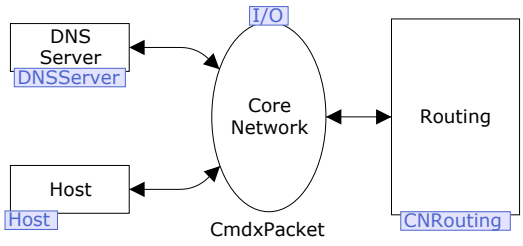


Fig. 24. Core Network module – modelling the core network.

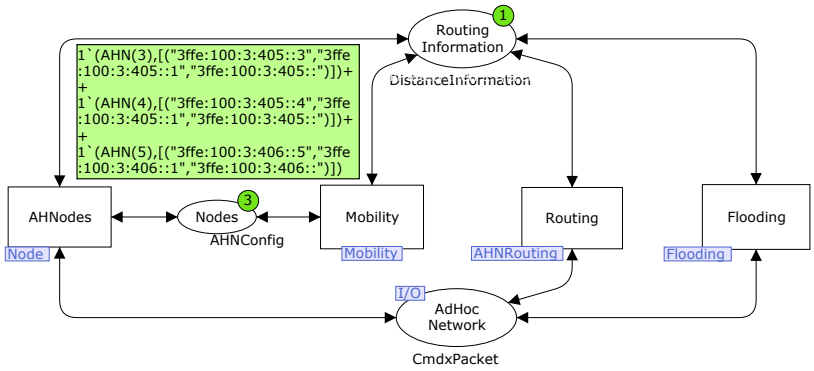


Fig. 25. AdHocNetwork module – modelling the ad-hoc network.

Nodes and the colour of the tokens specifies the name of the node and a list of configured IP addresses. Each configuration specifies the IP address configured, and the IP address and prefix of the corresponding gateway. It is possible for a mobile ad-hoc node to configure an IP address for multiple gateways. The mobile node must ensure that the DNS database always contains the IP address corresponding to the *preferred gateway*. In the marking shown in Fig. 25, it can be seen from the labels below the mobile nodes that Ad-hoc Node 3 and Ad-hoc Node 4 have configured IP addresses based on the prefix announced by Gateway1, whereas Ad-Hoc Node 5 has configured an IP address based on the prefix announced by Gateway2. For an example, Ad-hoc Node 3 has configured the address 3ffe:100:3:405::3.

```
(* --- ad-hoc nodes --- *)
color AHId = int with 1..5;
color AHNode = union AHN : AHId;

(* --- configuration information for ad-hoc nodes --- *)
color AHNIPConfig = product IPAdr * IPAdr * Prefix;
color AHNIPConfigs = list AHNIPConfig;
color AHNConfig = product AHNode * AHNIPConfigs;
```

Fig. 26. Colour definitions used in the AdHocNetwork module.

There are four substitution transitions in the AdHocNetwork module corresponding to the components of the ad-hoc network. The substitution transition AHNodes represents the behaviour of the nodes in the mobile ad-hoc network. The substitution transition Mobility models the mobility of nodes in the ad-hoc

network, i.e., that the nodes may move closer or further away from the gateways. The substitution transition Routing represents the routing protocol executed in the ad-hoc network. The purpose of the routing protocol in the context of the RIP protocol is to provide the nodes with information about distances to the gateways. The routing is abstractly modelled in a similar way as the routing mechanism in the core network and will not be discussed further in this paper. The substitution transition Flooding models the dissemination of advertisements from the gateways. A detailed presentation of this part of the model has been omitted here. The complete CPN model of the RIP protocol is hierarchically structured into 18 modules. A detailed presentation of the CPN model can be found in [74].

5.2 Behavioural Visualisation of the RIP Protocol

In the routing interoperability project, the BRITNeY Suite animation framework [111] was used to create an *animation GUI* on top of the CPN model. The animation GUI allows a user to observe the execution of the constructed CPN model using a graphical representation of the network architecture. The graphics is updated by the underlying CPN model according to the execution of the formally specified protocol, and the CPN model is also able to react to stimuli provided by the user via the animation GUI.

Figure 27 shows a representative snapshot of the application-specific graphics during the execution of the CPN model. The IP addresses configured by the individual nodes are shown as labels below the nodes. For an example, Ad-hoc Node 3 has configured two IP addresses: 3ffe:100:3:405:3 and 3ffe:100:3:406:3. The convention is that the preferred IP address is the topmost address in the list below the node. The entries in the DNS database are shown in the upper left corner. It shows the entries for each of the three ad-hoc nodes. The two numbers written at the top of each node are counters that provide information about the number of packets on the incoming (left) and outgoing (right) interfaces of the nodes. Transmissions of advertisements from the gateways are visualised by green dots. Fig. 27 shows an example where Gateway2 is transmitting an advertisement. Transmission of payload packets is visualised using red dots, and DNS packets are visualised using blue dots.

In addition to observing feedback on the execution of the CPN model in the animation GUI, it is also possible to provide input to the CPN model directly via the animation GUI. The user can move the nodes in the ad-hoc network thereby changing the distances to the two gateways. It is also possible to define a packet flow from the host in the core network to one of the nodes in the mobile ad-hoc network by clicking on the red square positioned next to each of the ad-hoc nodes. The square will change its colour to green once the CPN model has registered the flow. The flow can be stopped again by clicking on the (now green) square next to the mobile ad-hoc node. Finally, it is possible to force the transmission of an advertisement from a gateway by clicking on the gateway.

A more generic form of high-level graphical feedback in the form of MSCs was also used in this project. Figure 28 shows an example of an MSC diagram based

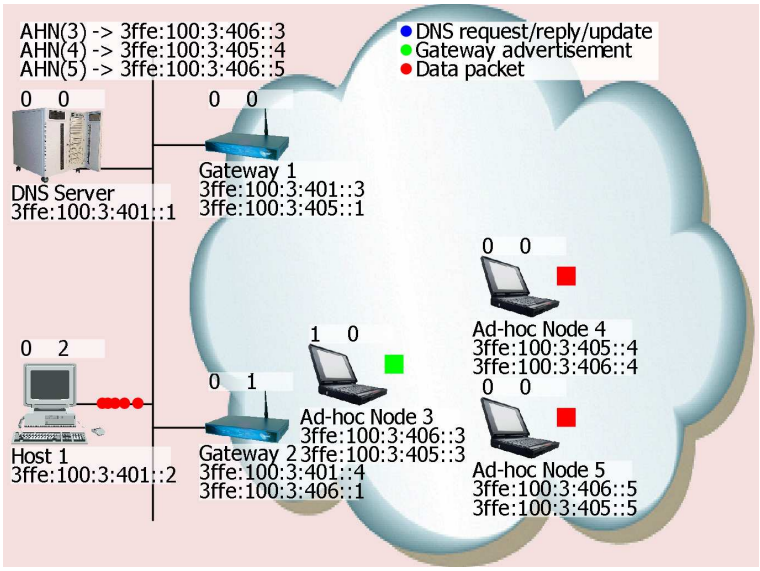


Fig. 27. Snapshot of the interaction graphics.

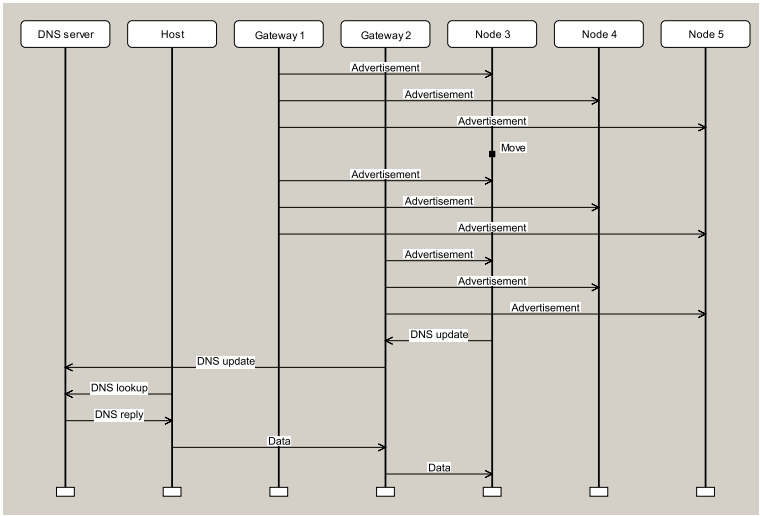


Fig. 28. Message sequence chart generated by the animation GUI.

on a simulation of the CPN model. The MSC shows a scenario where Ad-hoc Node 3 makes a Move and discovers that Gateway 2 is now the closest gateway. This causes it to send a DNS update to the DNS server. The last part of the MSC shows the host initiating a packet flow to Ad-hoc Node 3. One benefit of using MSCs is that they provide an event-based view that records the execution history. This is in contrast to the state-based view on the CPN model that one obtains during an interactive simulation. The two forms of feedback therefore complement each other and MSCs have been widely used in projects where CPNs were applied to protocol design.

Graphical feedback from the execution of the CPN model is achieved by attaching *code segments* to the transitions in the CPN model. These code segments are sequential pieces of code that are executed whenever the corresponding transition occurs in the simulation/execution of the CPN model. As an example consider the CNRouting module in Fig. 29. The transition Route models the routing of the packet on the core network. It uses the routing information on place RoutingInformation to direct the packet to the proper gateway. The SML function FindNextHop in the guard expression of the transition computes the IP address of the next hop gateway using the routing information and destination IP address of the packet. The Route transition has an attached code segment which is executed whenever the transition occurs. The code segment invokes the primitives in the animation package for animating the transmission of packets in the core network.

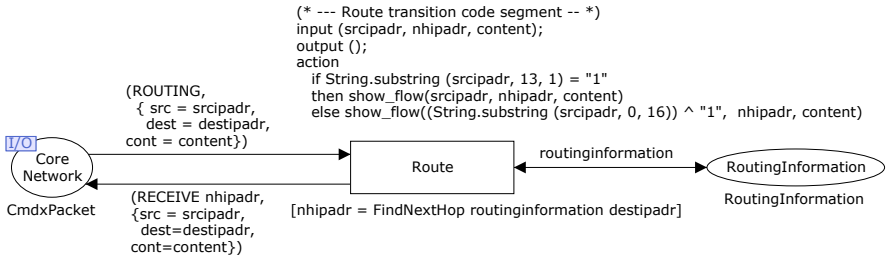


Fig. 29. The CNRouting module – Routing in the core network.

The CPN model receives input from the animation GUI by polling the animation GUI for events. An event queue has been implemented between the animation GUI and the CPN model. The code segment of transition Produce in the Poll module shown in Fig. 30 polls the animation GUI for events at regular intervals during the execution of the CPN model. Events are put into a list-token representing an event queue on the place Events. The parts of the CPN model that are to react on events from the animation GUI are linked via place fusion to the Event place and are able to consume events from the event queue. The occurrence of the transition Produce corresponds to a poll to the animation GUI for events.

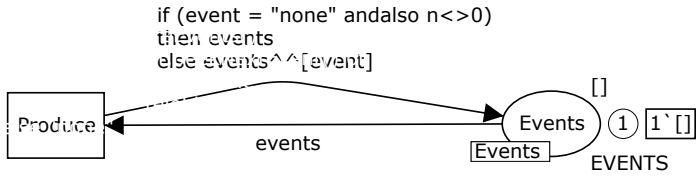


Fig. 30. The Poll module – Polling the animation GUI for events.

5.3 Lessons Learned and Perspectives

The CPN model combined with the animation GUI that was developed in the RIP project served as an early model-based executable prototype. The domain specific graphical user interface (the animation GUI) made it possible to explore and demonstrate the design of the interoperability protocol with the underlying formal model being transparent for the observer and the demonstrator. In particular, it made it possible for persons without knowledge of the CPN modelling language to experiment with the proposed design. The use of an animation GUI on top of the CPN model has the advantage that the behaviour observed by the user is as defined by the underlying model that formally specifies the design. The alternative would have been to implement a separate visualisation application totally detached from the CPN model. This would have led to double representation of the dynamics of the interoperability protocol which could in turn lead to inconsistencies between the two representation of the design.

Another advantage offered by the development of a model-based prototype is ease of control compared to a physical prototype, in particular in the case of mobile nodes and wireless communication where scenarios can be very difficult to control and reproduce. The use of a model means that there is no need to invest in physical equipment and there is no need to setup the actual physical equipment early in the project. The use of a model also makes it possible to investigate larger scenarios, e.g., scenarios that may not be feasible to investigate with the available physical equipment. An additional general advantage of the approach taken in the RIP project is that at an early stage of development, the implementation details can be abstracted away and only the key part of the design have to be specified in detail. As an example, the CPN model of the interoperability protocol abstracted away the routing mechanisms in the core and ad-hoc networks, and the mechanism used for distribution of advertisements. Instead, the service assumed from these components for the interoperability protocol to work was modelled. The possibility of making abstraction means that it is possible to obtain an executable prototype without implementing all the components.

6 The Edge Router Discovery Protocol

The previous sections have demonstrated how modelling, simulation, state space exploration, and behavioural visualisation can be applied for validating the functional design of protocols. This section summarises a project [67] conducted with Ericsson Telebit A/S where a combination of the techniques introduced in the previous sections were applied for the design of an Edge Router Discovery Protocol (ERDP). The CPN model of ERDP was developed in close cooperation with the protocol engineers at Ericsson Telebit A/S based on a natural-language specification that would normally have served as a basis for the implementation of the protocol. Simulation and MSCs were used in initial investigations of the ERDP protocol behaviour. Then state space exploration was used to conduct a formal verification of the key behavioural properties of ERDP. The aim of this section is to show how modelling, simulation, visualisation, and state space exploration all can help to identify omissions and behavioural errors in a design, and how they are typically used in conjunction in a protocol design process.

6.1 CPN Model of the ERDP Protocol

ERDP is based on the IPv6 Neighbour Discovery Protocol (NDP) [88] and supports *edge routers* residing on the boundary of an *IP core network* in configuring *gateways* with an IPv6 address prefix. This address prefix can in turn be used by mobile nodes in ad-hoc networks to configure global IPv6 unicast addresses and obtain Internet access via the core network. Figure 31 shows the ERDP module which is the top-level module of the CPN model. The substitution transition Gateway represents the gateway, and the substitution transition EdgeRouter represents the edge router. The wireless communication link between the edge router and the gateway is represented by the substitution transition GW_ER_Link. The four socket places GWIn, GWOut, ERIn, and EROut model packet buffers between the link layer and the gateway and edge router. Both the gateway (GW) and the edge router (ER) have an incoming and an outgoing packet buffer.

All four places in Fig. 31 have the colour set IPv6Packet, used to model the IPv6 packets exchanged between the edge routers and gateways. Since ERDP is based on the IPv6 Neighbour Discovery Protocol, the packets are carried as Internet Control Message Protocol (ICMP) packets. The definitions of the colour sets for NDP, ICMP, and IPv6 packets were derived directly from RFC 2460 [22] by using record type constructors for representing fields within packets and union type constructors for representing the different kinds of packets (see [67] for detail). It was considered important by the protocol engineers for later implementation that the definition of the packets followed closely the structure of IPv6 packets instead of a more abstract representation.

Figure 32 shows the EdgeRouter module. The port places ERIn and EROut are related to the accordingly named socket places in the ERDP module (see Fig. 31). The place Config models the configuration information associated with the edge router, and the place PrefixCount models the number of prefixes still available in

the edge router for distribution to gateways. The place PrefixAssigned is used to keep track of which prefixes are assigned to which gateways.

Figure 33 shows the declarations of the colour sets for the three places in Fig. 32. The configuration information for the edge router (modelled by the colour set ERConfig) is a record consisting of the IPv6 link-local address and the link-layer address of the edge router. A list of pairs (colour set ERPrefixAssigned) consisting of a link-local address and a prefix is used to keep track of which prefixes are assigned to which gateways. A counter modelled by the place PrefixCount with the colour set PrefixCount is used to keep track of the number of prefixes still available. When this counter reaches 0, the edge router has no further prefixes available for distribution. The number of available prefixes can be modified by changing the initial marking of the place PrefixCount, which is set to 1 by default.

The substitution transition SendUnsolicitedRA (in Fig. 32) corresponds to the multicasting of periodic *unsolicited router advertisements* (RAs) by the edge router such that gateways can discover the presence of the edge router. When a gateway receives an unsolicited RA, it responds with a unicast *router solicitation* (RS). The substitution transition ProcessRS models the reception at the edge router of unicasted RSs from gateways, and the sending of a unicast RA to the gateway in response. The substitution transition ERDiscardPrefixes models the expiration of prefixes on the edge router side.

The marking shown in Fig. 32 has a single token on each of the three places used to model the internal state of the edge router protocol entity. In the marking shown, the token on the place PrefixAssigned with the colour `[]` corresponds to the edge router not having assigned any prefixes to the gateways. The token on the place PrefixCount with colour `1` indicates that the edge router has a single prefix available for distribution. Finally, the colour of the token on the place Config specifies the link-local and link addresses of the edge router. In this case

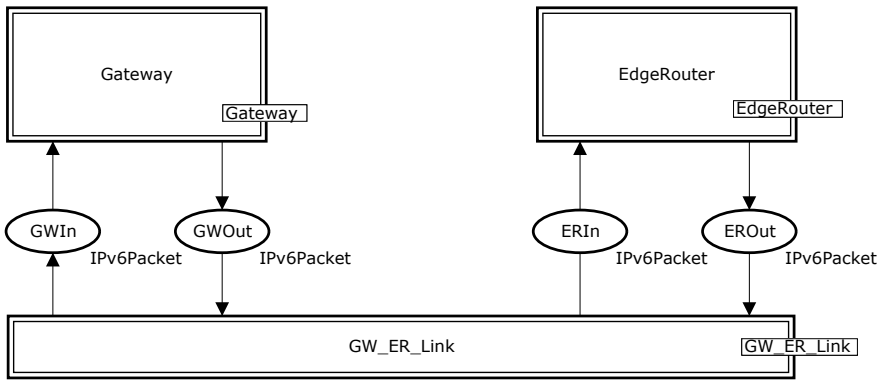


Fig. 31. Top-level module of the ERDP CPN model.

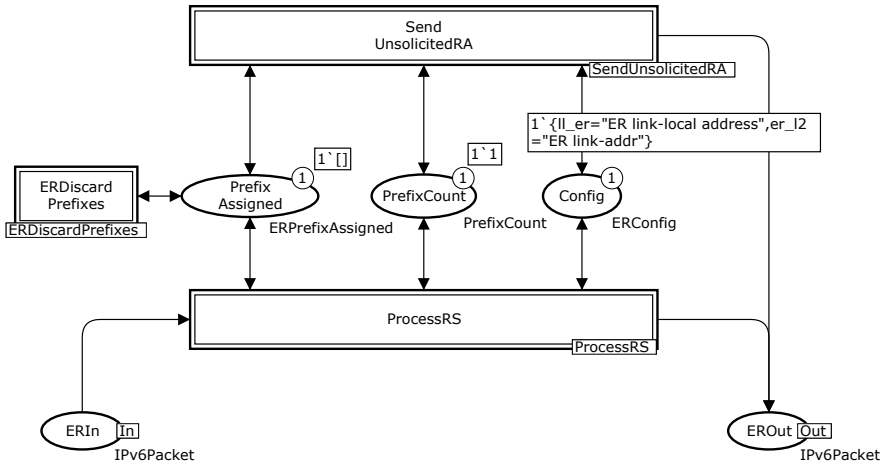


Fig. 32. The EdgeRouter module.

```
colset LinkAddr    = string;

colset ERConfig = record ll_er : IPv6Addr * (* link-local address  *)
                         er_l2 : LinkAddr;  (* link-addr (layer 2) *)

colset ERPrefixEntry    = product IPv6Addr * IPv6Prefix;
colset ERPrefixAssigned = list ERPrefixEntry;

colset PrefixCount = int;
```

Fig. 33. Colour set definitions for edge routers.

the edge router has the symbolic link-local address ER link-local address, and the symbolic link-address ER link-addr.

Figure 34 depicts the SendUnsolicitedRA module which is the submodule of the substitution transition SendUnsolicitedRA in Fig. 32. The transition SendUnsolicitedRA models the sending of the periodic unsolicited router advertisements. The variable erconfig is of type ERConfig, and the variable prefixleft is of type PrefixCount. The transition SendUnsolicitedRA is enabled only if the edge router has prefixes available for distribution, i.e., prefixleft is greater than 0. This is ensured by the function SendUnsolicitedRA in the guard of the transition.

Figure 35 depicts the marking of the SendUnsolicitedRA module after the occurrence of the transition SendUnsolicitedRA in the marking shown in Fig. 34. An unsolicited router advertisement has been put in the outgoing buffer of the edge router. It can be seen that the `DestinationAddress` is the address

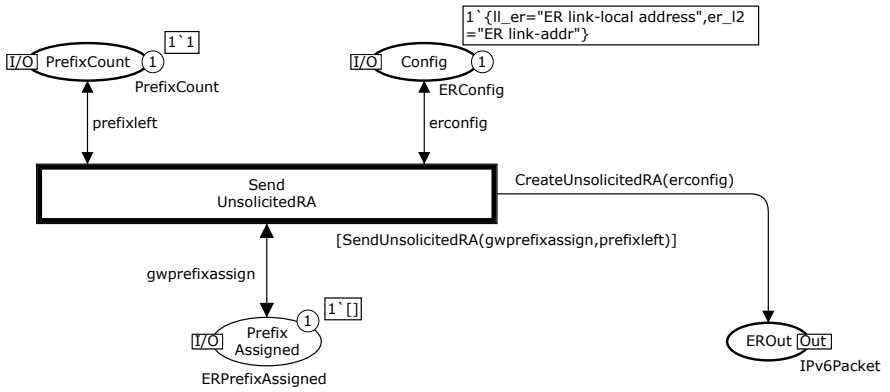


Fig. 34. Initial marking of the SendUnsolicitedRA module.

`all-nodes-multicast`, the `SourceAddress` is `ER link-local address`, and the `LinkLayerAddress` (in the options part) is `ER link-addr`.

Figure 36 shows the part of the GW_ER_Link module that models transmission of packets from the edge router to the gateway across the wireless link. Transmission of packets from the gateway to the edge router is modelled similarly. The port places GWIn and EROut are linked to the similarly named socket places in Fig. 31. The transition ERtoGW models the successful transmission of packets, whereas the transition LossERtoGW models the loss of packets. The variable ipv6packet is of type IPv6Packet. A successful transmission of a packet

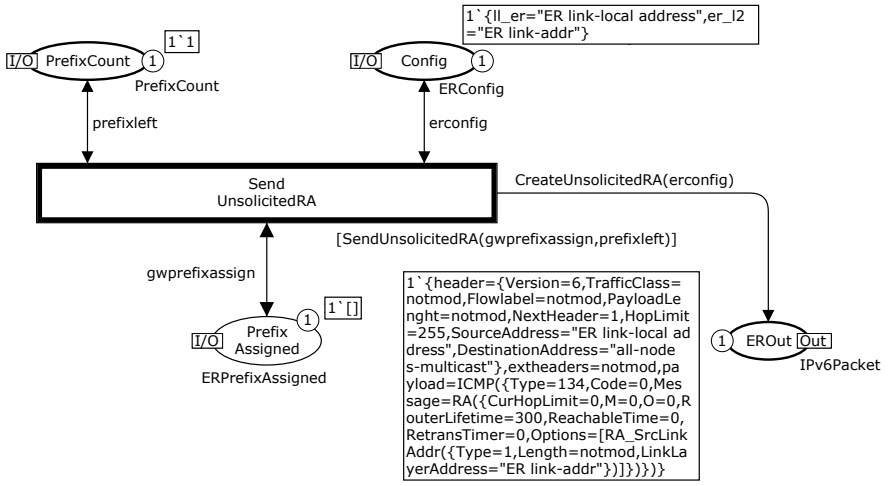


Fig. 35. Module SendUnsolicitedRA, after occurrence of SendUnsolicitedRA.

from the edge router to the gateway corresponds to moving the token modelling the packet from the place EROut to GWIn. If the packet is lost, the token will only be removed from the place EROut.

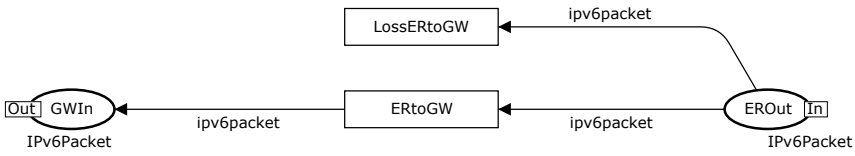


Fig. 36. Part of the GW_ER_Link module.

Wireless links, in general, have a lower bandwidth and higher error rate than wired links. These characteristics have been abstracted away in the CPN model since the purpose is to reason not about the performance of ERDP but rather its logical correctness. Duplication and reordering of messages are not possible on typical one-hop wireless links, since the detection of duplicates and the preservation of order are handled by the data-link layer. The modelling of the wireless links does allow overtaking of packets, but this overtaking was eliminated in the state space exploration phase where bounds were imposed on the capacity of the input and output packet buffers.

The CPN model was developed as an integrated part of the development of ERDP. The creation of the CPN model was done in cooperation with the protocol engineers at Ericsson in parallel with the development of the ERDP specification. Altogether 70 person-hours were spent on CPN modelling. Prior to the development of the CPN model, the protocol engineers at Ericsson were given a 6 hour course on CPNs that made them capable of reading CPN models. This means that CPN models could be used actively in discussion related to the design of the ERDP protocol. MSCs (to be illustrated shortly), integrated with simulation were used in both review steps to investigate the behaviour of ERDP in detail. The use of MSCs in the project was of particular relevance since it presented the operation of the protocol in a form well known to the protocol engineers. Altogether 24 design issues were identified during three iterations on the CPN model. Table 2 categorises and enumerates the issues encountered during two review phases (Review 1 and Review 2) of the protocol design. The issues were identified in the process of constructing the CPN model, performing single-step executions of the CPN model, and engaging in discussions of the CPN model with the protocol engineers at Ericsson.

Table 2. ERDP project [67] – design issues identified in the modelling phase.

Category	Review 1	Review 2	Total
Errors in protocol specification/operation	2	7	9 issues
Incompleteness and ambiguity in specification	3	6	9 issues
Simplifications of protocol operation	2	0	2 issues
Additions to the protocol operation	4	0	4 issues
Total	11	13	24 issues

6.2 Verification of the ERDP CPN Model

State space exploration was conducted after the three iterations of modelling as discussed in the previous section. The purpose of the state space exploration was to conduct a more thorough investigation of the operation of ERDP, including verification of its key properties. The key behavioural property of ERDP is proper configuration of the gateway with prefixes. This means that for a given prefix and state where the gateway has not yet been configured with that prefix, the protocol must be able to configure the gateway with that prefix. Furthermore, when the gateway has been configured with the prefix, the edge router and the gateway should be *consistently configured*, i.e., the assignment of the prefix must be recorded both in the gateway and in the edge router protocol entity. Whether a marking represents a consistently configured state for a given prefix can be checked by inspecting the marking of the place PrefixAssigned in the edge router and the marking of the place Prefixes in the gateway.

Obtaining a finite state space. The first step towards state space exploration of the CPN model was to obtain a finite state space. The CPN model as presented above has an infinite state space, since an arbitrary number of tokens (packets) can be put on the places modelling the packet buffers. As an example, the edge router may initially send an arbitrary number of unsolicited router advertisements. To obtain a finite state space, an upper integer bound of 1 was imposed on each of the places GWIn, GWOut, ERIn, and EROut (see Fig. 31) which model the packet buffers. This also prevents overtaking among the packets transmitted across the wireless link. Furthermore, the number of packets simultaneously present in the four input/output buffers was limited to 2. Technically, this was done by using the *branching options* available in the CPN state space tool to prevent the processing of enabled transitions whose occurrence in a given marking would violate the imposed bounds on the buffer places.

No packet loss and prefix expire. The second step was to consider the simplest possible configurations of ERDP, starting with a single prefix and assuming that there is no packet loss on the wireless link and that prefixes do not expire. The

full state space for this configuration had 46 nodes and 65 arcs. Inspection of the state space report showed that there was a single dead marking represented by node 36. Inspection of this node showed that it represented a state where all of the packet buffers were empty. However, the edge router and gateway were inconsistently configured in this state in that the edge router had assigned the prefix P1 (the single prefix), while the gateway was not configured with that prefix. This was an error in the protocol. To locate the source of the problem, query functions in the state space tool were used to obtain a counter example leading from the node representing the initial marking to node 36. Figure 37 shows the resulting error trace, visualised by means of an MSC. This MSC was generated automatically from the extracted counter example. The column labelled GW-Buffer represents the packet buffer between the gateway protocol entity and the underlying protocol layers. Similarly, the ERBuffer column represents the packet buffer in the edge router. The problem is that the edge router sends two unsolicited RAs. The first one gets through and the gateway is configured with the prefix, which can be seen from the event marked with *A* in the lower part of the MSC. However, when the second RS, without any prefixes, is received by the edge router (the event marked with *B*), the corresponding solicited RA will not contain any prefixes. Because of the way the protocol was specified, the gateway will therefore update its list of prefixes to the empty list (the event marked with *C*), and the gateway is no longer configured with a prefix.

To fix the error, the protocol was modified so that the edge router always replies with the list of all prefixes that it has currently assigned to the gateway. The state space for the modified protocol consisted of 34 nodes and 49 arcs, and there were no dead markings in the state space. The state space report specified that there were 11 home markings. Inspection of these 11 markings showed that they all represented consistently configured states for the prefix P1. The markings were contained in the single terminal SCC of the state space. A terminal SCC is an SCC of the state space where all successors of states in the SCC belong to the SCC itself. This shows that, from the initial marking it is always possible to reach a consistently configured state for the prefix, and that when such a marking has been reached, the protocol entities will remain in a consistently configured state. To verify that a consistently configured state would eventually be reached, it was checked that the single terminal SCC was the only non-trivial SCC. A trivial SCC is a SCC consisting of just a single state. This showed that all cycles in the state space (which correspond to non-terminating executions of the protocol) were contained in the single terminal SCC, which (from above) contained only consistently configured states. The reason why the protocol is not supposed to terminate in a consistently configured state represented by a dead marking is that the gateway may, at any time, when it is configured, send a router solicitation back to the edge router to have its prefixes refreshed.

Increasing the number of prefixes. When the correctness of the protocol had been established for a single prefix, the number of prefixes was increased. When there is more than one prefix available it no longer holds that a marking will *eventually*

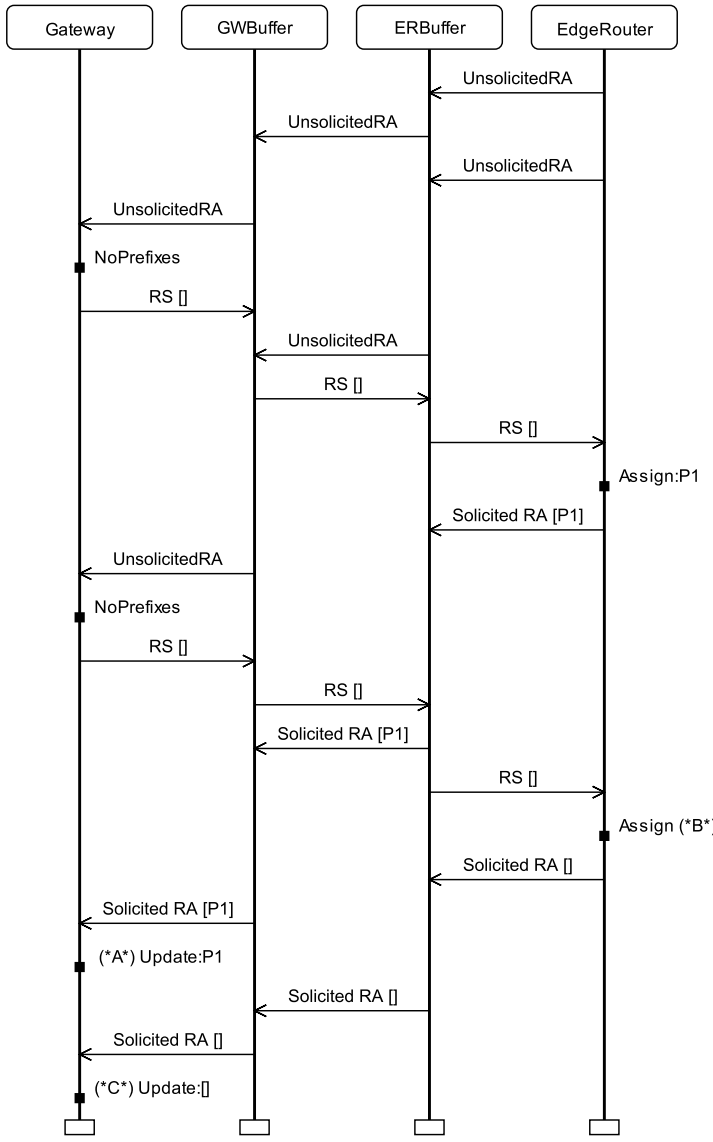


Fig. 37. MSC showing an execution leading to an undesired terminal state.

be reached where *all* prefixes are consistently configured. The reason is that with more than one prefix, the edge router may at any time decide not to configure the gateway with additional prefixes. Hence, a state where all prefixes have been consistently configured might not eventually be reached. Instead, firstly, it was verified that there was a single terminal SCC, all markings of which represent states where all prefixes have been consistently configured. This shows that it is always possible to reach such a marking, and when the protocol has consistently configured all prefixes, the protocol entities will remain consistently configured. Secondly, it was checked that all markings in each non-trivial SCC represented markings where the protocol entities were consistently configured with a subset of the prefixes available in the edge router. The properties above was checked using a number of user-defined queries in the state space tool of CPN Tools.

Adding packet loss. The third step was to allow packet loss on the wireless link between the edge router and the gateway. First, the case was considered in which there is only a single prefix for distribution. The state space for this configuration had 40 nodes and 81 arcs. Inspection of the state space report showed that there was a single dead marking. This marking represented an undesired terminal state where the prefix had been assigned by the edge router, but the gateway was not configured with the prefix. The source of the problem was located by extracting a counter example and visualising it in a similar manner as shown in Fig. 37. The problem was fixed by ensuring that the edge router would resend an unsolicited RA to the gateway as long as it had prefixes assigned to the gateway. The state space of the revised CPN model had 68 nodes and 160 arcs. Inspection of the state space report showed that there were no dead markings and no home markings. Investigation of the terminal SCCs showed that there were two terminal SCCs, each containing 20 markings. The nodes in one of them all represented states where the edge router and gateway were consistently configured with the single prefix P1, whereas the nodes in the other terminal SCC all represented states where the protocol entities were not consistently configured. The markings in the undesired terminal SCC represent a livelock in the protocol, i.e., if one of the markings in the undesired terminal SCC is reached, it is no longer possible to reach a state where the protocol entities are consistently configured with the prefix. The source of the livelock was related to the control fields used in the router advertisements for refreshing prefixes and their interpretation on the gateway. This was identified by obtaining the MSC for a path leading from the initial marking to one of the markings in the undesired terminal SCC. As a result, the processing of router advertisements in the gateway was modified. The state space for the protocol with the modified processing of router advertisements also had 68 nodes and 160 arcs. The state space had a single terminal SCC containing 20 nodes, which all represented states where the protocol entities were consistently configured with the single prefix.

When packet loss is present, it is not immediately possible to verify that the two protocol entities will eventually be consistently configured. The reason is that any number of packets can be lost on the wireless link. Each of the

non-trivial SCCs was inspected using a user-defined query to investigate the circumstances under which the protocol entities would not eventually be consistently configured. This query checked that either all nodes in the non-trivial SCC represented consistently configured states or none of the nodes in the SCC represented a consistently configured state. For those non-trivial SCCs where no node represented a consistently configured state, it was checked that all cycles contained the occurrence of a transition corresponding to loss of a packet. Since this was the case, it can be concluded that any failure to reach a consistently configured states will be due to packet loss only. Hence, if finitely many packets are lost, a consistently configured state for some prefix will *eventually* be reached.

Adding prefix expire. The fourth and final step in the analysis was to allow prefixes to expire. The analysis was conducted first for a configuration where the edge router had only a single prefix to distribute. The state space for this configuration had 173 nodes and 513 arcs. The state space had a single dead marking, and inspection of this dead marking showed that it represented a state where the edge router has no further prefixes to distribute, it has no prefixes recorded for the gateway, and the gateway is not configured with any prefix. This marking is a desired terminating state of the protocol, as we expect prefixes to eventually expire. Since the edge router has only finitely many prefixes to distribute, the protocol should eventually terminate in such a state. The single dead marking was also a home marking, meaning that the protocol can always enter the expected terminal state. When prefixes can expire, it is possible that the two protocol entities may never enter a consistently configured state. The reason is that a prefix may expire in the edge router (although this is unlikely) before the gateway has been successfully configured with that prefix. Hence, we are only able to prove that for any marking where a prefix is still available in the edge router, it is possible to reach a marking where the gateway and the edge router are consistently configured with that prefix.

Table 3 lists statistics for the size of the state space in the three verification steps for different numbers of prefixes. The column '|P|' specifies the number of prefixes. The columns 'Nodes' and 'Arcs' give the numbers of nodes and arcs, respectively, in the state space. For the state spaces obtained in the first verification step, it can be seen that 38 markings and 72 arcs are added for each additional prefix. The reason for this is that ERDP proceeds in phases where the edge router assigns prefixes to the gateway one at a time. Configuring the gateway with an additional prefix follows exactly the same procedure as that for the assignment of the first prefix. Once the state space had been generated, the verification of properties could be done in a few seconds. It is also worth observing that as the assumptions are relaxed, i.e., we move from one verification step to the next, the sizes of the state spaces grow. This, combined with the identification of errors in the protocol even in the simplest configuration, without packet loss and without expiration of prefixes, shows the benefit of starting state space exploration from the simplest configuration and gradually lifting

assumptions. Furthermore, the state explosion problem was not encountered during the verification of ERDP, and the key properties of ERDP were verified for the number of prefixes that were envisioned to appear in practise.

Table 3. State space statistics for the three verification steps.

	No loss/No expire		Loss/No Expire		Loss/Expire	
|P|	Nodes	Arcs	Nodes	Arcs	Nodes	Arcs
1	34	49	68	160	173	531
2	72	121	172	425	714	2 404
3	110	193	337	851	2 147	7 562
4	148	265	582	1 489	5 390	19 516
5	186	337	926	2 390	11 907	43 976
6	224	409	1 388	3 605	23 905	89 654
7	262	481	1 987	5 185	44 550	169 169
8	300	553	2 742	7 181	78 211	300 072
9	338	625	3 672	9 644	130 732	505 992
10	376	697	4 796	12 625	209 732	817 903

6.3 Lessons Learned and Perspectives

The project at Ericsson highlights the benefits of a formal modelling and validation approach. Furthermore, the project emphasised the benefits of the model construction phase which is often underestimated (or not reported) in literature on protocol validation. As illustrated by the ERDP project, the modelling phase itself lead to significant insight into the protocol design, and contributed to a simpler and more complete protocol design. The construction of a CPN model and subsequent state space exploration can be seen as a very thorough and systematic way of reviewing the ERDP design specification. The project showed that the process of constructing a CPN model based on the ERDP specification provided valuable input to the ERDP design, and the use of simulation added further insight into the operation of the protocol. State space exploration, starting with the simplest possible configuration of the protocol, identified additional errors in the protocol. The results from state space exploration also demonstrate that errors are often present in the smallest configurations of a protocol system.

Using an iterative process where both a conventional natural-language specification and a CPN model were developed (as in this project) turned out to be an effective way of integrating CPN modelling and validation into the development of a protocol. In general, the combination of an executable formal model (such as a CPN model) and a natural-language specification seems to be provide a useful way to develop a protocol. One reason why both are required is that the

software engineers that are eventually going to implement the protocol (which may be different from those that design the protocol) in many cases will not be familiar with the CPN modelling language. Secondly, in many cases there are important implementation elements of the protocol specification that are not reflected in the CPN model, such as the layout of packets.

It can be argued whether or not the issues and errors discovered in the process of modelling and conducting state space exploration would have been identified if additional conventional reviews of the ERDP specification had been conducted. Some of them probably would have been, but more subtle problems such as the inconsistent configurations discovered during state space exploration would probably not have been discovered until the first implementation of ERDP was operational. The reason for this is that discovering these problems requires one to consider subtle execution sequences of the protocol.

Overall, the application of CPNs in the development of ERDP was considered a success for three main reasons. Firstly, it was demonstrated that the CPN modelling language and supporting computer tools were powerful enough to specify and verify a real-world protocol being developed in an industrial project, and that integration into the conventional protocol development process is not difficult. Secondly, the act of constructing the CPN model, executing it, and discussing it led to the identification of several non-trivial design errors and issues that, under normal circumstances, would not have been discovered until, at best, the implementation phase. Finally, the effort of constructing the CPN model and conducting state space exploration was represented by approximately 100 person-hours. This is a relatively small investment compared with the many issues that were identified and resolved early as a consequence of constructing and analysing the CPN model.

7 Related Work on CPN Protocol Validation

The four protocol examples presented in this paper constitute only a small subset of the examples that have been published in the literature on the use of CPNs for specification and validation of protocols - in particular in relation to protocols developed in the context of IETF and other protocol standardisation bodies.

The Datagram Congestion Control Protocol (DCCP) developed by the IETF has been investigated in [11]. DCCP is intended to provide an unreliable transport service with congestion control mechanisms. The work in [11] was done in parallel with the development of the emerging DCCP standard, and concentrated on modelling and verification of the connection establishment and synchronisation procedures of DCCP. It resulted in the identification of several functional errors in the protocol design, including discovery of deadlocks, non-progress behaviour (chatter), and problems with connection establishment in relation to sequence number wraps. The formal validation resulted in the IETF working group making small (but important) changes to the connection establishment and synchronisation procedures of DCCP. The work also included the development of a formal service specification for DCCP [33] and application of the

sweep-line method [105] for on-the-fly checking of the protocol conformance to the developed service specification.

The classical Transmission Control Protocol (TCP) has also been modelled and verified using CPNs [10]. Similar to the work on DCCP, this work concentrated on the connection establishment procedures. It resulted in verifying the absence of deadlocks and livelocks in connection establishment, and a detailed specification of the circumstances under which TCP connection establishment may not be successful. Another example of transport layer protocol modelling and validation can be found in [104] which considers the Stream Transmission Control Protocol (SCTP).

The Internet Open Trading Protocol (IOTP) designed to provide an interoperability framework for Internet commerce was formally modelled and validated using CPNs in [92, 91, 90]. IOTP is designed to handle common trading procedures and encompass trading roles such as consumer, merchant, payment handler, and delivery handler. IOTP is organised around a collection of eight baseline transactions consisting of Purchase, Refund, Value exchange, Authentication, Withdrawal, Deposit, Inquiry, and Ping. These transactions comprise a minimal set of transactions for an Internet commerce protocol. A formal specification of the service provided by IOTP was developed using CPN in [92]. The service was specified in the form of a finite-state automaton labelled with service primitives. The automaton was extracted from the state space of the CPN model by identifying the binding elements corresponding to service primitives of the protocol. A CPN model of the IOTP protocol itself was presented in [91, 90]. State space exploration focused on the termination properties and absence of livelocks in the IOTP transactions. The use of state space exploration revealed deficiencies related to termination of transactions. A verification of the IOTP protocol CPN model [91, 90] against the formal service specification from [92] was presented in [89]. Finite-state automata language comparison was used as the criterion for conformance following the methodology of [9]. Application of the sweep-line state space method on IOTP was investigated in [34] exploiting an inherent progression from the start of an IOTP transaction to termination of the transaction.

The Wireless Application Protocol (WAP) has been considered in [40, 41]. WAP is designed to provide Internet services to a wide range of hand-held devices. The work of [40, 41] concentrates on the Wireless Transaction Protocol (WTP) which is an important element of the WAP architecture and protocol suite. The work in [40] presents a formal modelling of the WTP service and a formal modelling of the WTP protocol. Checking the conformance of the WTP protocol against the WTP service was done using finite-state automata language comparison. This approach succeeded in detecting several inconsistencies between the protocol and the service which was provided as input to the WAP forum responsible for the development of WAP. The sweep-line method was used in [41] to alleviate the state explosion problem and allow for the verification of larger configurations of WTP. The application of the sweep-line method allowed

configurations with parameter settings of retransmission counters corresponding to the recommended setting for GSM and IP network to be verified.

The Session Initiation Protocol (SIP) is a widely used protocol for the establishment of Internet multimedia session, and has been subject to formal modelling and validation in [77, 23]. The INVITE transactions have been formally analysed using state space exploration in [77, 23] leading to identification of undesired terminating states of the protocol when operating over an unreliable communication medium. Security aspects of SIP have been investigated in [78]. The work of [37] focuses on the formal modelling of a SIP-based protocol for multichannel service oriented architectures. A formalisation of SIP with the purpose of providing a framework model for present architectures in mobile computing is presented in [36]. Another multimedia control protocol, the Capability Exchange Signalling (CES) protocol, has been formally modelled using CPNs and verified using state space exploration in [79]. The work on the CES protocol led to the identification of protocol errors in presence of sequence number wrap. Suggested changes were incorporated in a revised CPN model, and it was formally verified showing that the discovered errors have been eliminated.

The NEO protocol which is part of the distributed transactional object database management system NEOPPOD was investigated using high-level Petri Nets in [17]. The Coloane environment was used for the construction of the models, and verification was performed using the CPN-AMI and Helena tools. The NEO protocol is used to coordinate data storage and retrieval in a decentralised and distributed system where data can be stored on a number of data nodes and data is accessed through the primary master node. The focus of [17] was on the protocol used for the election of the primary master node. The model of the election part of the NEO protocol consisted of eighteen modules. Since there existed no specification document for the protocol, the Petri net model was reverse-engineered from a prototype implementation. The validation process which relied on the use of state spaces discovered two flaws in the implementation of the protocol. These were subsequently provided to the software engineers responsible for the implementation of the protocol component.

The Resource Reservation Protocol (RSVP) was formally modelled and verified in [107, 106]. The modelling and verification concentrates on verifying the absence of deadlocks and livelocks in relation to the setup, maintenance and path release procedures of RSVP. In addition, a number of RSVP specific behavioural properties were investigated which considered in detail the internal state of the sender, router, and receiver protocol entities of the protocol. The main contribution of the work was the development of a formal specification of the RSVP path procedures. Another example on the modelling of routing protocols can be found in [76] which uses Mobile Petri Nets to construct a formal model of the Mobile IP protocol. Mobile IP allows transport layer connections to be preserved when mobile nodes change their point of attachment to the Internet. CPNs have also recently been used for the verification of security protocols. Privacy enhancing protocols were considered in [99], and [39] addresses the modelling and validation of PANA Authentication and Authorisation Protocol. Examples of protocols for

which parametric verification has been pursued in the context of CPNs can be found in [32, 31].

8 Conclusions and Outlook

Functional validation of protocol designs is one of the main application areas of CPNs and supporting computer tools [28]. In this paper, we have surveyed a selection of recent projects on modelling and functional validation of industry relevant protocols. The examples demonstrate how the elements of protocols can be modelled using CPNs, and they illustrate how a combination of simulation, application-specific behavioural visualisation, and state space exploration is typically applied in protocol validation with CPNs. From a modelling perspective, the protocol examples have ranged from models representing two (or few) peer protocol entities (e.g., GAN, EDRP, and RIP) having an explicit representation in the net structure, to parameterised models capable of modelling an arbitrary number of peer protocol entities by setting a model parameter (e.g., DYMO). The latter was based on constructing a folded model where the identity of the protocol entities is encoded explicitly as part of the token colours. The CPN models have also illustrated modelling at different protocols layer ranging from models operating at a single protocol layer (e.g., DYMO and ERDP) to protocol system design involving multiple protocol layers and protocols (e.g., GAN and RIP). An important aspect of the examples is that the process of modelling and conducting single step simulation is an important (but often underestimated) activity in the validation of a protocol design.

The main technique available for functional verification of CPN models is that of explicit state space exploration. The examples presented in this paper show how basic state space exploration combined with the generation of a state space report relying on a number of standard behavioural properties of Petri Nets, provides a light-weight approach which in many cases is an important step in verifying key properties of a protocol design. The main reason for the wide spread application of state space exploration has been the presence of mature computer tool support combined with the main advantages of state space exploration in terms of being a highly systematic approach, being able to provide counter examples, and allowing for a high degree of automation. The compact modelling of protocols enabled by CPNs has, in many cases, had the effect that the full state space can be explored for at least the smallest configuration of the considered protocol. The GAN and ERDP examples presented in this paper are concrete examples illustrating this. Practise have shown that the primary capability offered by the advanced state space methods is the possibility of verifying larger configurations of the protocol - and in some cases [71] the configurations of the system that are expected to occur in practise. The ERDP example considered in this paper is another example of this. Hence, despite the fact that explicit state space exploration methods requires one to conduct verification relative to a particular configuration of the protocol, the current suite of availably

state space methods combined with the power of modern computing platforms in many situations allows for the practical validation of industrial-sized protocols.

While CPNs have been successfully applied to modelling and validating protocol designs, there has been relatively few attempts at using the constructed CPN models in an automated or semi-automated manner as a basis for the actual implementation of protocols. Some simulation-based approaches were used in [87] and [70] for generating server-side implementations. Here, the simulation code for the CPN model generated by CPN Tools was extracted, and after undergoing automatic modifications (e.g., linking the code to external libraries), the generated simulation code is used as the system implementation. A limitation of this approach is that the execution speed is affected because each step in the execution of the program involves the computation and execution of enabled transitions (as done by a CPN simulator) in order to determine the next state. Secondly, the approach ties the target platform to that of the CPN Tools simulator which may make the approach impractical for many application domains due to resource consumption of the CPN simulator. The SML/NJ compiler used for the simulator in CPN Tools has a large memory footprint making it ill-suited, e.g., for the domain of embedded systems. Some initial work on a translation-based approach can be found in [73]. Here a restricted form of CPNs was used for obtaining an Erlang implementation of the DYMO routing protocol. The approach in [73] relies on the use of Process-Partitioned CPNs which enforces a detailed modelling of the protocol design which is very close to an implementation level model. An area that will be important as part of efforts in developing capabilities for automated code generation is the development of CPN protocol modelling methodology on which only limited research has been undertaken [18].

Acknowledgements. The authors acknowledge the contribution of Kristian L. Espensen and Mads. K. Kjeldsen in the project on the DYMO protocol, Paul Fleischer for his contribution in the GAN project, Michael Westergaard and Peder Christian Nørgaard for their contribution to the project on the RIP protocol, and Kurt Jensen for his contributions in the ERDP project.

References

1. ISO/IEC 15437. Information technology. Enhancements to LOTOS (E-LOTOS), September 2001.
2. 3GPP. Digital Cellular Telecommunications System (Phase 2+); Generic Access to the A/Gb Interface; Stage 2. 3GPP TS 43.318 version 6.9.0 Release 6, March 2007.
3. 3GPP. Website of 3GPP. `http://www.3gpp.org`, May 2007.
4. R. Alur, G. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.
5. M. A. Ardis. Formal Methods for Telecommunication System Requirements: A Survey of Standardised Languages. *Annals of Software Engineering*, 3, 1997.
6. C. Baier and J-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
7. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *LNCS*, 2004.

8. J. Billington, G. Gallasch, L.M. Kristensen, and T. Mailund. Exploiting Equivalence Reduction and the Sweep-Line Method for Detecting Terminal States. *IEEE Transactions on Systems, Man, and Cybernetics. Part A: Systems and Humans*, 34(1):23–38, 2004.
9. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 210–290. Springer, 2004.
10. J. Billington and B. Han. Modelling and Analysing the Functional Behaviour of TCPs Connection Management Procedures. *International Journal on Software Tools for Technology Transfer*, 9(3-4):269–304, 2007.
11. J. Billington and S. Vanit-Anunchai. Coloured Petri Net Modelling of an Evolving Internet Protocol Standard: The Datagram Congestion Control Protocol. *Fundamenta Informaticae*, 88(3):357–385, 2008.
12. J. Billington and C. Yuan. On Modelling and Analysing the Dynamic MANET On-Demand (DYMO) Routing Protocol. In *Transactions on Petri Nets and Other Models of Concurrency*, volume 5800 of *LNCS*, pages 98–126, 2009.
13. G. Bochmann. Finite state description of protocols. *Computer Networks*, pages 361–372, 1978.
14. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14:25–59, 1987.
15. I.D. Chakeres and C.E. Perkins. Dynamic MANET On-demand (DYMO) Routing. `http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-10.txt`, July 2007. Internet-Draft. Work in Progress.
16. I.D. Chakeres and C.E. Perkins. Dynamic MANET On-demand (DYMO) Routing. `http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-11.txt`, November 2007. Internet-Draft. Work in Progress.
17. C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification. In *Proc. of ICATPN'10*, volume 6128 of *LNCS*, pages 145–164. Springer, 2010.
18. C. Choppy, L. Petrucci, and G. Reggio. A Modelling Approach with Coloured Petri Nets. In *Proc. of 13th International Conference on Reliable Software Technologies*, volume 5026 of *LNCS*, pages 73–86, 2008.
19. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.
20. E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. *Formal Methods in System Design*, 9:77–104, 1996.
21. D. E. Comer. *Internetworking with TCP/IP Volume 1: Principles, Protocols, and Architecture*. Prentice-Hall, 5th edition, 2005.
22. S. Deering and R. Hinden. Internet Protocol, Version 6 (IPV6) Specification, December 1998. RFC 2460.
23. L.G. Ding and L. Liu. Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets. In *Proc. of ICATPN'08*, volume 5062 of *LNCS*, pages 132–151. Springer, 2008.
24. E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9:105 – 131, 1996.
25. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *Proc. of ICATPN'08*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.

26. ETSI. ETSI ES 201 873-1: Methods for Testing and Specification; The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.
27. S. Evangelista, M.Westergaard, and L.M. Kristensen. The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. *Transactions on Petri Nets and Other Models of Concurrency*, 3:189–215, 2009.
28. Examples of Industrial Use of CPNs. `http://cs.au.dk/cpnets/industrial-use/`.
29. A. Fehnker, R. van Glabbeek, P. Hofner, A. McIver, M. Portmann, and W. Tan. Modelling and Analysis of AODV in UPPAAL. In *Proc. of 1st Workshop on Rigorous Protocol Engineering*, 2011.
30. P. Fleischer and L.M. Kristensen. Modelling and Validation of Secure Connection Establishment in a Generic Access Network Scenario. *Fundamenta Informaticae*, 94(3-4):361–386, 2009.
31. G. E. Gallasch and J. Billington. Using Parametric Automata for the Verification of the Stop and Wait Class of Protocols. In *Proc 3rd Int. Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *LNCS*, pages 457–473. Springer, 2005.
32. G. E. Gallasch and J. Billington. A Parametric State Space for the Analysis of the Infinite Class of Stop-and-Wait Protocols. In *Proc. of SPIN Workshop on Model Checking of Software*, volume 3925 of *LNCS*, pages 201–218. Springer, 2006.
33. G. E. Gallasch, J. Billington, S. Vanit-Anunchai, and L.M. Kristensen. Checking Safety Properties On-the-fly with the Sweep-Line Method. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):371–392, 2007.
34. G.E. Gallasch, C. Ouyang, J. Billington, and L.M. Kristensen. Experimenting with Progress Mappings for the Sweep-Line Analysis of the Internet Open Trading Protocol. In *Proc. of 5th Workshop and Tutorial on Practical Use of Coloured Petri Nets and CPN Tools (CPN'04)*, pages 19–38, 2004.
35. Guy Edward Gallasch, Bing Han, and Jonathan Billington. Sweep-Line Analysis of TCP Connection Management. In *ICFEM*, volume 3785 of *LNCS*, pages 156–172. Springer, 2005.
36. V. Gehlot and A. Hayrapetyan. A Formalized and Validated Executable Model of the SIP-based Presence Protocol for Mobile Applications. In *Proceedings of the 45th Annual ACM Southeast Regional Conference*, pages 185–190. ACM, 2007.
37. Vijay Gehlot and Anush Hayrapetyan. A CPN Model of a SIP-Based Dynamic Discovery Protocol for Webservices in a Mobile Environment. In *Proc. of 7th Workshop and Tutorial onPractical Use of Coloured Petri Nets and the CPN Tools (CPN'06)*, 2006.
38. B. Genest, A. Muscholl, and D. Peled. Message sequence charts. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 537–558. Springer, 2004.
39. S. Gordon. Formal Analysis of PANA Authentication and Authorisation Protocol. In *Proc. of 9th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 277–284. IEEE Computer Society, 2008.
40. S. Gordon and J. Billington. Analysing the WAP Class 2 Wireless Transaction Protocol Using Coloured Petri Nets. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 207–226. Springer, 2000.
41. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer, 2002.
42. P. Grimstrup. Interworking Description for IKEv2 Library. Ericsson Internal. Document No. 155 10-FCP 101 4328 Uen, September 2006.

43. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.
44. C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
45. G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
46. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
47. The Internet Engineering Task Force (IETF). `http://www.ietf.org`.
48. C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9:41 – 75, 1996.
49. ISO9074. Information Processing Systems - Open Systems Interconnection: ESTELLE (FOrmal Description Technique Based on an Extended State Transition Model.
50. ISO89 ISO/IEC. Information Processing Systems - Open Systems Interconnection: LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, February 1989. IS 8807.
51. ITU-T. Z.120: Message Sequence Charts (MSC), 1996.
52. ITU-T. Z.109: SDL-2000 Combined with UML, 2000.
53. ITU-T. X.680 to X.683: Abstract Syntax Notation One, 2002.
54. ITU-T. X.692 - Encoding Control Notation, 2002.
55. ITU-T. Z.100-Z.106: Specification and Description Language (SDL), 2010.
56. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1:Basic Concepts*. Monographs in Theoretical Computer Science. Springer, 1992.
57. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2: Analysis Methods*. Monographs in Theoretical Computer Science. Springer, 1994.
58. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9:7 – 40, 1996.
59. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
60. K. Jensen, L.M. Kristensen, and T. Mailund. The sweep-line state space exploration method. *Theoretical Computer Science*, 429:169–179, 2012.
61. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.
62. J.B. Jørgensen and L.M. Kristensen. Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):714–732, 1999.
63. J.B. Jørgensen and L.M. Kristensen. Verification of Coloured Petri Nets Using State Spaces with Equivalence Classes. In *Petri Net Approaches for Modelling and Validation*, volume 1 of *LINCOM Studies in Computer Science*, chapter 2, pages 17–34. Lincoln Europa, 2003.
64. C. Kaufman. Internet Key Exchange Protocol. RFC 4306, December 2005.
65. S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, December 2005.
66. L.M. Kristensen. A Perspective on Explicit State Space Exploration of Coloured Petri Nets: Past, Present, and Future. In *Proc. of ICATPN'10*, volume 6128 of *LNCS*, pages 39–42. Springer, 2010.

67. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Prot ocol for Mobile Ad-hoc Networks. In *Proc. of Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.
68. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proc. of FME'02*, volume 2391 of *LNCS*, pages 549–567. Springer, 2002.
69. L.M. Kristensen and T. Mailund. Efficient Path Finding with the Sweep-Line Method Using External Storage. In *Proc. of ICFEM'03*, volume 2885 of *LNCS*, pages 319–337. Springer, 2003.
70. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G. E. Gallasch. Model-based Development of COAST. *STTT*, 10(1):5–14, 2007.
71. L.M. Kristensen, J.B. J ørgensen, and K. Jensen. Application of Coloured Petri Nets in System Development. In *Proc. of 4th Advanced Course on Petri Nets Lectures on Concurrency and Petri Nets - Advances in Petri Nets*, volume 3098 of *LNCS*, pages 626–685. Springer, 2004.
72. L.M. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In *Proc. of ICATPN'98*, volume 1420 of *LNCS*, pages 104–123. Springer, 1998.
73. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'2010*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
74. L.M. Kristensen, M. Westergaard, and P.C. Nrgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of Fifth International Conference on Integrated Formal Methods (IFM05)*, volume 3771 of *LNCS*, pages 266–286. Springer, 2005.
75. R. Lai and A. Jirachiefpattana. *Communication Protocol Specification and Verification*. Kluwer Academic Publishers, 1998.
76. C. Lakos. Modelling Mobile IP with Mobile Petri Nets. *Transactions on Petri Nets and Other Models of Concurrency*, 5800:127–158, 2009.
77. L. Liu. Verification of the SIP Transaction Using Coloured Petri Nets. In Bernard Mans, editor, *Thirty-Second Australasian Computer Science Conference (ACSC 2009)*, volume 91 of *CRPIT*, pages 63–72, Wellington, New Zealand, 2009. ACS.
78. L. Liu. Security Analysis of Session Initiation Protocol - A Methodology Based on Coloured Petri Nets. In *Proc. of the 2010 International Cyber Resilience Conference*, 2010.
79. L. Liu and J. Billington. Verification of the Capability Exchange Signalling protocol. *STTT*, 9(3-4):305–326, 2007.
80. Ming T. Liu. Protocol Engineering. *Advances in Computers*, 29:79–195, 1989.
81. L. Lorentsen and L.M. Kristensen. Modelling and Analysis of a Danfoss Flowmeter System Using Coloured Petri Nets. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 346–366. Springer, 2000.
82. IETF Mobile Ad-hoc Networks Discussion Archive. `http://www1.ietf.org/mail-archive/web/manet/current/index.html`.
83. T. Mailund. Analysing Infinite-State Systems by Combining Equivalence Reduction and the Sweep-Line Method. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 314–333. Springer, 2002.
84. R. Malik and R. Mühlfeld. A case study in verification of uml statecharts: the profisafe protocol. *Universal Computer Science*, 9(2):138–151, 2003.
85. G. Malkin. RIP Version 2. RFC 4822, February 2007.

86. R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
87. K.H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 367–386. Springer, 2000.
88. T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6), December 1998. RFC 2461.
89. C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proc. of 4th International EC-Web Conference*, volume 2738 of *LNCS*, pages 292–302. Springer, 2003.
90. C. Ouyang and J. Billington. Formal Analysis of the Internet Open Trading Protocol. In *Proc. of Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops*, volume 3236 of *LNCS*, pages 1–15. Springer, 2004.
91. C. Ouyang, L.M. Kristensen, and J. Billington. A Formal and Executable Specification of the Internet Open Trading Protocol. In *Proc. of Third International Conference on E-Commerce and Web Technologies*, volume 2455 of *LNCS*, pages 377–387. Springer, 2002.
92. C. Ouyang, L.M. Kristensen, and J. Billington. A Formal Service Specification for the Internet Open Trading Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 352–373. Springer, 2002.
93. C.A. Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
94. M. Popovic. *Communication Protocol Engineering*. CRC Press, 2006.
95. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ICATPN'03*, volume 2679 of *LNCS*, pages 450–462. Springer, 2003. `http://www.cpntools.org`.
96. A. P. Ravn, J. Srba, and S. Vighio. Modelling and verification of web services business activity protocol. In *Proc. of TACAS'11*, volume 6605 of *Lecture Notes in Computer Science*, pages 357–371, 2011.
97. W. Reisig. *Petri Nets - An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
98. U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Proc. of Correct Hardware Design and Verification Methods*, volume 987 of *LNCS*, pages 206–224. Springer, 1995.
99. S. Suriadi, C. Ouyang, J. Smith, and E. Foo. Modeling and Verification of Privacy Enhancing Protocols. In *Proc. of ICFEM'09*, volume 5885 of *LNCS*, pages 127–146. Springer, 2009.
100. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
101. A. Valmari. A Stubborn Attack on State Explosion. In *Proc. of Computer-Aided Verification (CAV'90)*, volume 531 of *LNCS*, pages 156–165. Springer, 1990.
102. A. Valmari. Stubborn Sets of Coloured Petri Nets. In *Proc. of ICATPN'91*, pages 102–121, 1991.
103. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.
104. S. Vanit-Anunchai. Towards Formal Modelling and Analysis of SCTP Connection Management. In *Proc. of CPN'09*, pages 163–182, 2008.
105. S. Vanit-Anunchai, J. Billington, and G. E. Gallasch. Analysis of the Datagram Congestion Control Protocols Connection Management Procedures Using the Sweep-line Method. *International Journal on Software Tools for Technology Transfer*, 10(1):29–56, 2008.

106. M.E. Villapol and J. Billington. A Coloured Petri Net Approach to Formalising and Analysing the Resource Reservation Protocol. *CLEI Electron. J.*, 6(1), 2003.
107. M.E. Villapol and J. Billington. Analysing Properties of the Resource Reservation Protocol. In *Proc. of ICATPN'03*, volume 2679 of *LNCS*, pages 377–396. Springer, 2003.
108. P. Vixie. Dynamic Updates in the Domain Name System. RFC 2136, April 1997.
109. M. Westergaard, S. Evangelista, and L.M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *Proc. of ICATPN'09*, volume 5606 of *LNCS*, pages 303–312. Springer, 2009. `http://www.daimi.au.dk/~ascoveco/download.html`.
110. M. Westergaard, L.M. Kristensen, G.S. Brodal, and L.A. Arge. The ComBack Method – Extending Hash Compaction with Backtracking. In *Proc. of ICATPN'07*, volume 4546 of *LNCS*, pages 445–464. Springer, 2007.
111. M. Westergaard and K. B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ICATPN'06*, volume 4024 of *LNCS*, pages 431–440. Springer, 2006.
112. Michael Westergaard. A Game-theoretic Approach to Behavioural Visualisation. *Electr. Notes Theor. Comput. Sci.*, 208:113–129, 2008.
113. P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer, 1993.

Chapter 8

Towards a CPN-based Modelling Approach for Reconciling Verification and Implementation of Protocol Models

Towards a CPN-based Modelling Approach for Reconciling Verification and Implementation of Protocol Models

Kent Inge Fagerland Simonsen[1,2] and Lars Michael Kristensen[1]

[1] Department of Computer Engineering, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
[2] DTU Informatics, Technical University of Denmark, Denmark
Email: kisi@imm.dtu.dk

Abstract Formal modelling of protocols is often aimed at one specific purpose such as verification or automatically generating an implementation. This leads to models that are useful for one purpose, but not for others. Being able to derive models for verification and implementation from a single model is beneficial both in terms of reduced total modelling effort and confidence that the verification results are valid also for the implementation model. In this paper we introduce the concept of a descriptive specification model and an approach based on refining a descriptive model to target both verification and implementation. Our approach has been developed in the context of the Coloured Petri Nets (CPNs) modelling language. We illustrate our approach by presenting a descriptive specification model of the Websocket protocol which is currently under development by the Internet Engineering Task Force (IETF), and we show how this model can be refined to target both verification and implementation.

Keywords: Protocol software for pervavise computing, model-based protocol development, protocol verification, Coloured Petri Nets.

1 Introduction

Common uses of protocol modelling are to describe, to specify, to verify, and to generate implementations of protocol software. These uses are typically not supported by a single model, although all of these objectives can be achieved using a single modelling language. For example, in order to conduct state space-based verification of a Petri Net model of a protocol, it is a requirement that the state-space of the model is of a size that can be represented given the available computing resources. However, to be able to automatically generate executable code, a model needs to include many details that significantly increase the size of the state space (or even make it infinite). In order to describe a protocol, the model should be at a level of abstraction that provides enough detail to understand the concepts and operation of the protocol, but should not include the

abstractions that arise from the need of limiting the state space nor the details needed to generate an implementation of the protocol for a specific platform.

Coloured Petri Nets (CPNs) [12] have been widely used to verify protocols (e.g., [6,14,16]) and associated verification methodologies [2] has been developed. Examples also exists (e.g., [15]) where CPN models have been developed with the purpose of obtaining implementations. Limited work [4] exists on methodologies for CPN modelling of protocol software, and earlier work has not attempted to identify and make explicit the differences and commonalities between verification and implementation models. Also, there are comparatively few examples in the literature where a CPN model has been constructed with the purpose of describing a protocol. One contribution of this paper is to present a modelling approach based on the concept of a *descriptive specification model* which serves as a common origin model for deriving *verification* and *implementation models*. By abstractions and restrictions of the scope of the model, a descriptive model can be transformed into a model suited for verification. The descriptive model can also be transformed into a model suited for implementation via the addition of *code generation pragmatics* and by means of refinement. To illustrate our modelling approach, we present a descriptive specification model of the Websocket (WS) protocol [8] currently under development by IETF. The Websocket protocol provides a message-oriented full-duplex connection and relies on the Transmission Control Protocol (TCP) and the Hypertext Transfer Protocol (HTTP). The WS protocol targets web applications and uses HTTP to open a connection. For the data transfer, the WS protocol relies directly on bidirectional TCP streams in order to avoid the request-response (polling) pattern of HTTP, and to eliminate the overhead that would be induced by the verbose HTTP headers. Data framing is used on top of the TCP streams to make the WS connections message-oriented. Based on the descriptive model of the WS protocol, we derive and present verification and implementation models. The WS protocol has (to the best of our knowledge) not been subject to formal modelling and verification which constitute the second contribution of this paper.

CPNs have several useful features that makes it a good choice as a modelling language for our approach. The concurrency model inherent in CPNs makes it easy to make realistic models of protocol systems where several principals are executing concurrently. Also, since CPNs are executable, simulation makes it easy to visualise the flow of the modelled system. In addition, the hierarchical structure of CPNs is highly amenable to structuring an entire protocol system from the principals and channels at the highest level through the service declarations to the specific operations of a service. A main difference between CPNs and other modelling language for protocol systems like the the Extended State Transition Language (Estelle) [9] and the Specification and Description Language (SDL) [11] is that CPNs have very few (but still powerful) modelling constructs. Estelle and SDL have a large and complex set of language constructs to describe the behaviour of protocol principals and their interaction. From this perspective, CPNs provide a simpler and more lightweight approach to protocol modelling which at the same time is less implementation specific than, e.g., typical SDL

protocol specifications. In that respect, CPNs are close to languages like the Language of Temporal Ordering Specification (LOTOS) [1] that focus on abstract and implementation independent protocol specification. In a UML context, state diagrams (charts) [7] have been used widely for modelling protocol modules and message sequence charts (MSCs) [10] (sequence diagrams in UML) are being used in particular for specifying protocols requirements that can later be used in protocol verification [5]. MSCs have also been used for protocol specification using higher-level control flow constructs. In contrast to MSCs which are action-oriented, state charts and CPNs are both state and action-oriented modelling formalisms. The above observations combined with our existing expertise and experience made us choose CPNs as the underlying modelling language in our approach. With the basis of our approach established, it is likely that future work will include the creation of a domain specific language on top of CPNs to model protocols.

The rest of this paper is organised as follows: Section 2 presents the characteristics of descriptive specification models and the descriptive CPN model of the WS protocol. Section 3 discusses how a verification model can be derived from a descriptive model, and presents an initial verification of the WS protocol using the obtained verification model. Section 4 shows how the descriptive WS model can be transformed into an implementation model. Finally, Sect. 5 contains the conclusions and a discussion of future work. The reader is assumed to be familiar with the basic concepts of Petri Nets and explicit state model checking. Due to space limitations, we give only a very limited presentation of CPN concepts. The reader is referred to [12,13] for an introduction to the CPN modelling language.

2 A Specification Model of the Websocket Protocol

A descriptive specification model, in our modelling approach, is a model that has as its primary purpose to facilitate description and communication of how a protocol works, and to serve as an executable protocol specification that provides a basis for verification and implementation models. The descriptive aspect of models tends to receive less focus in the academic modelling literature even though understanding and communicating how software works are considered by developers to be important reasons to create models and sketches [3]. The descriptive CPN model for the WS protocol presented in this section has been created based on the WS RFC specification document [8] using CPN Tools [20]. The focus of the descriptive model is on the logical operation of the protocol which implies that the model does not encompass quantitative properties such as timing constraints and resource usage. It is possible to add such aspect in a refined model using the time concept provided by CPNs. In the following, we describe selected parts of the constructed CPN model. The complete model can be found via [17].

One specific feature of a descriptive model is to be able to show the operation of the protocol at different levels of abstraction from the protocol architecture through the major components down to the specific component behaviour. This

is important both for understanding the protocol as a whole as well as allowing different stake-holders to focus on the appropriate levels of abstraction. Another important feature of a descriptive model is a high level of readability. This means that it should be easy for human readers to read and understand the model and, with the help of the model, understand the protocol. What makes a model readable and whether the example presented in this paper fits this descriptions will be the subject of further investigations. A descriptive model should also include all the important parts of the protocol as well as all the major states the protocol may be in. This is important to ensure that the model can be used as a basis for deriving implementation and verification models and also such that the descriptive specification model can be used to understand and communicate the operation of all the major parts of the protocol.

To create the descriptive model of the WS protocol, the first step is to create sub-modules for each of the *principals* (protocol entities) and the *communication channels* between them on the top module of the model. This is useful for explaining the overall architecture of the protocol. After the high-level architecture has been modelled the next level includes the main components and states of each principal. The states and main components are identified by using the protocol specification document. This level is used to give a general overview of each of the principals. The subsequent levels should describe the components of each principal focusing on keeping as close to the protocol specification document as possible. With CPNs this is achieved by using a hierarchical model. Figure 1 shows the top-level module of the CPN model which reflects the overall architecture of the WS protocol. The WS protocol involves two principal actors: one client and one server which are represented by the *substitution transitions* *Client* and *Server*, respectively.

The two principals are connected by places representing channels from *ClientToServer* and from *ServerToClient*. Both places have the colour set `Connection` that is being used to model the TCP connection on top of which the WS connection is being established. The *colour set* `Connection` determining the kind of *tokens* that can reside on the two channel places (places in the CPN model that model communication channels) are defined in Fig. 2.

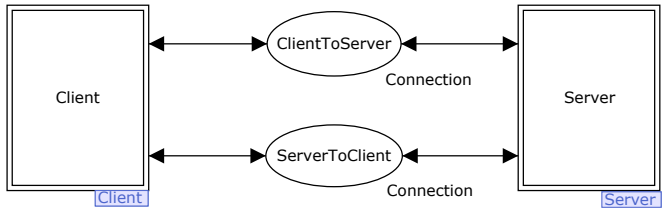


Figure 1. The top-level module of the descriptive specification model

```
colset Channel = with TCP;
colset Content = list Data;
colset Connection = product Channel * Content;
```

Figure 2. Colour set definition for channels and connections

The `Connection` colour set is a product type where the first component (`Channel`) specifies the type of the channel to be used for communication, and the second component (`Content`) is used to model the data currently in transmission on the channel. We discuss the `Data` colour set later.

In the descriptive model, the TCP connection is assumed to be a perfect stream connection. This means that all data that are sent from one endpoint will be received at the other endpoint, in the correct order and without alterations. It is also assumed that all data received from one endpoint are sent from the other endpoint and not injected into the channel through some other means. Finally, it is assumed that the channel is not closed abruptly. These *assumptions* are stronger than what TCP can guarantee, but are in accordance with the implicit assumptions on TCP made in the WS specification document [8]. Furthermore, the assumptions allow us to focus on the core behaviour of the protocol without considering all the possible errors that might originate from the platform that the protocol will eventually be executed on.

A major goal of our approach is to create a model with a one-to-one relationship between the concepts in the model and the specification document. When the model and the specification fails to have a one-to-one relationship, the reason for the discrepancy should be explained and documented. Discrepancies should only be tolerated when following the one-to-one correspondence would have a significant negative impact on the readability of the model and when breaking it does not significantly change the function of the protocol. Using the WS protocol over the Transport Layer Security (TLS) is left out of the scope of the WS descriptive model, and the same applies to error handling. These elements could be added to the model and would contribute to making the model more complete, but they would not provide insight in terms of illustrating our modelling approach. In addition, we have also left out optional extensions of the protocol in the modelling. When developing a model with a close relationship to the specification document using the method outlined above, it is our contention that the end result will be a model that is suitable for describing the protocol.

The main states that the principals participating in the WS protocol can be in are shown in Fig. 3 and Fig. 4 which depict the server and client submodules of the CPN model describing the WS protocol. In the initial state, *READY*, the client (and server) has not had any interaction. Once the WS connection has been established (substitution transitions *EstablishWebSocketConnection*), the principals enter the *OPEN* state. In the *OPEN* state, data transfer (substitution transition *DataTransfer*) can take place, until either of principals chooses to close

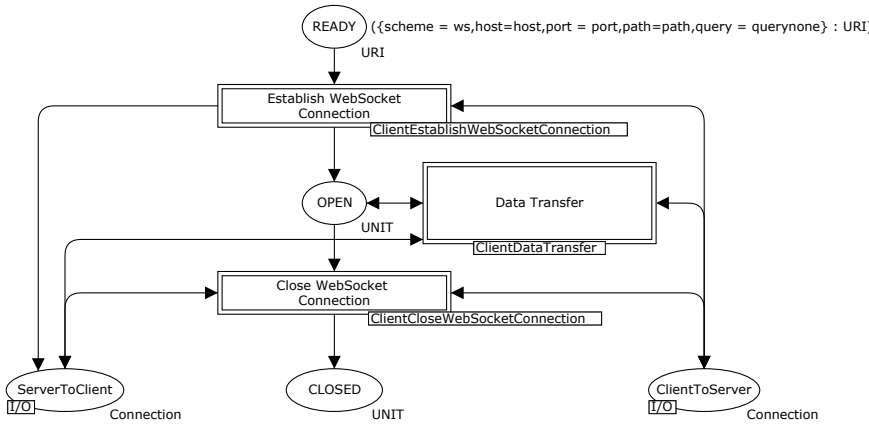


Figure 3. The top level module of the client principal

the connection (substitution transitions *CloseWebSocketConnection*). After the websocket connection has been closed, the principals enter a *CLOSED* state, and no further communications is possible.

The WS protocol relies on the use of HTTP *messages* for establishing the WS connection, and *frames* specifically designed for the WS protocol in order to provide a message-oriented channel on top of TCP streams. A central ab-

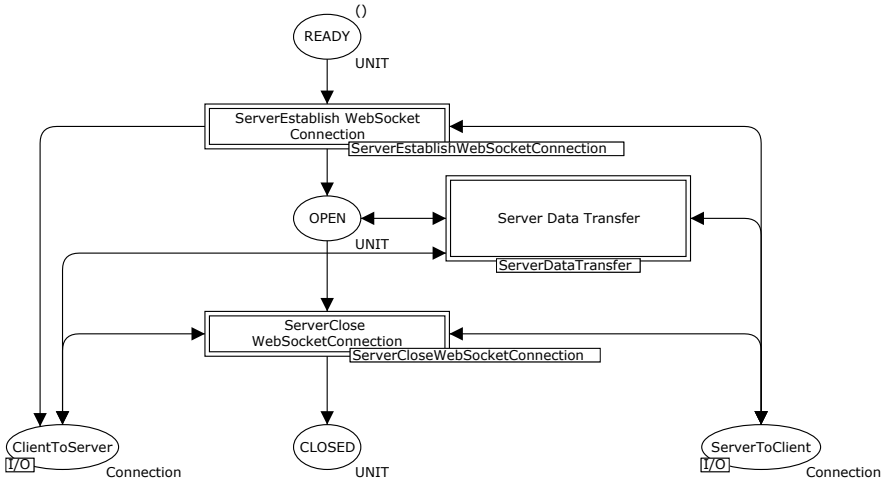


Figure 4. The top level module of the server principal

straction in the descriptive model is that we model the individual fields in the messages and frames, but abstract from the byte layout. Figure 5 lists the definitions of the colour sets (data types) that model the data exchanged between the principals. A number of the fields of frames and messages have been modelled using the `UNIT` colour set containing just the single value `()` (unit). The `UNIT` colour set is used in the cases where we have abstracted from the specific values of the fields while still modelling that the fields has to be represented in the corresponding message/frame. In particular, it can be seen that we have abstracted from the specific payload (and payload length) of frames as this does not affect the operation of the protocol, but only the users of the WS protocol. The `opcode` field of `Frame` is used to specify whether the frame is a data (text, binary or continuation), a ping, a pong or a close frame.

```
colset Data = union WSFRAME : Frame + HTTPREQ : HTTPRequest +
                    HTTPRESP : HTTPResponse;

colset Frame = record FIN  : BIT         * Opcode    : INT      *
                      MASK : BIT        * PayloadLen : UNIT *
                      Maskingkey : UNIT * PayloadData : UNIT;

colset HTTPRequest = record method   : HTTPMethod *
                            resource : Resource   *
                            version  : STRING     *
                            host     : Host       *
                            upgrade  : STRING     *
                            connection      : STRING *
                            secwebsocketkey : UNIT   *
                            secwebsocketversion : STRING;

colset HTTPResponse = record version : STRING * status : INT *
                             statusdescription  : STRING *
                             upgrade            : STRING *
                             connection         : STRING *
                             secwebsocketaccept : UNIT;
```

Figure 5. Colour set definition of HTTP messages and frames

2.1 Opening the Websocket Connection

The establishment of a WS connection is referred to as the *open handshake* and is modelled by the modules shown in Fig. 6 (client side) and Fig. 7 (server side). The first step is to open a TCP connection. In the descriptive model, we

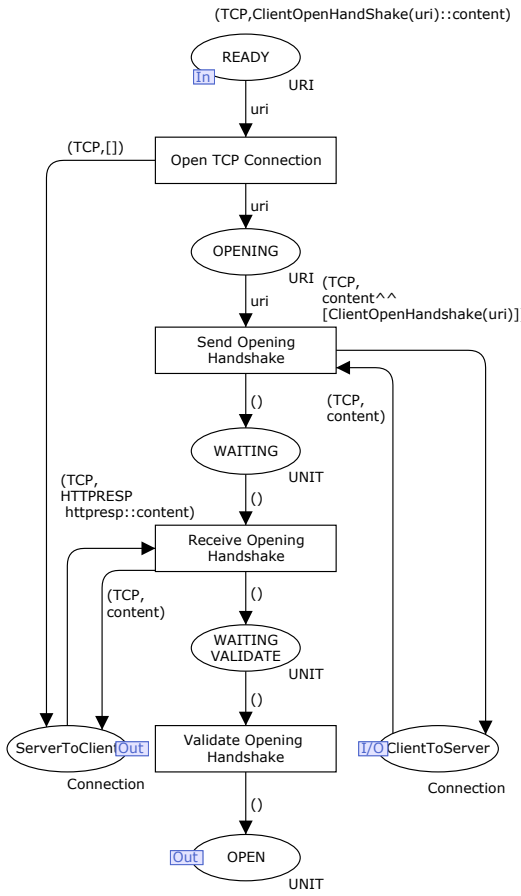


Figure 6. The Websocket open handshake - client side

have not explicitly modelled TCP. Instead, the TCP connection is modelled as being open when there is a token present on the channel place. The opening of the TCP connection (in the client to server direction) is modelled by the *OpenTCPConnection* transition in Fig. 6. In response to the client opening the TCP connection, the server opens the TCP connection in the server to client direction as modelled by the *AcceptTCPConnection* transition in Fig. 7.

When both sides have opened the TCP connection, the client sends its HTTP upgrade request. The request is created by the function shown in Fig. 8. The `method` field of a HTTPREQ message specifies that a GET operation is to be performed and the `uri` is used to identify the endpoint of the WS connection. The `host` field contains the hostname of the server. The `upgrade` and `connection` fields indicate that this is an upgrade request for a WS connec-

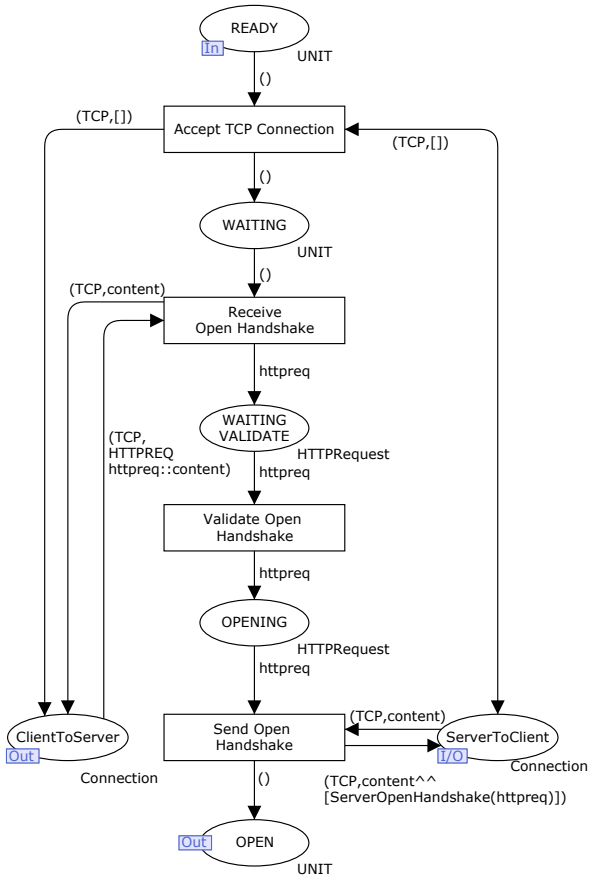


Figure 7. The Websocket open handshake - server side

tion. The `secwebsocketkey` field contains a base-64 encoded 16-byte nonce, but this type is in the model abstracted to a unit value. The `secwebsocketversion` indicates the version of the web socket protocol to be used.

Figure 9 shows the function used to create the HTTP response on the server side of the open handshake. The first field of the response is in the form of a status line in an HTTP response, which should include the status `101 Switching Protocols`. The rest of the response header fields follows the standard format for HTTP header fields. As with the Websocket upgrade request, the `upgrade` and `connection` fields indicate that the connection will be upgraded to a WS connection. The `secwebsocketaccept` header field contains the transformed value of the `secwebsocketkey` from the upgrade request.

After the client receives and validates the response from the server (modelled by the `ValidateOpeningHandshake` transition in Fig. 6), the client enters the

```
fun ClientOpenHandshake (uri:URI) =
 HTTPREQ {
   method  = GET,          resource   = #path uri,
   version = HTTP_VERSION, host       = #host uri,
   upgrade = UPGRADE,      connection = CONNECTION,
   secwebsocketkey     = (),
   secwebsocketversion = WEBSOCKETVERSION
  };
```

Figure 8. Client open handshake function

OPEN state. At this point both the client and the server are ready to send and receive frames and the open handshake is finished.

The websocket key in the model which is used to verify the handshake, and is included in the handshake messages as a unit value. Furthermore, the transformation and validation of this key is not included in the model, other than as a transition on both the client and server side named *ValidateOpenHandshake* that is always successful.

```
fun ClientOpenHandshake (uri:URI) =
 HTTPREQ {
   method  = GET,          resource   = #path uri,
   version = HTTP_VERSION, host       = #host uri,
   upgrade = UPGRADE,      connection = CONNECTION,
   secwebsocketkey     = (),
   secwebsocketversion = WEBSOCKETVERSION
  };
```

Figure 9. Server open handshake function

2.2 Data Transfer

Once both the client and server are in the *OPEN* state they may transmit data until they send or receive a close frame. In addition they may send ping and pong frames (to check that the connection is still alive). The module modelling the data transfer phase is shown in Fig. 10. The corresponding module for the server is symmetrical to the client data transfer model. Since the WS connection

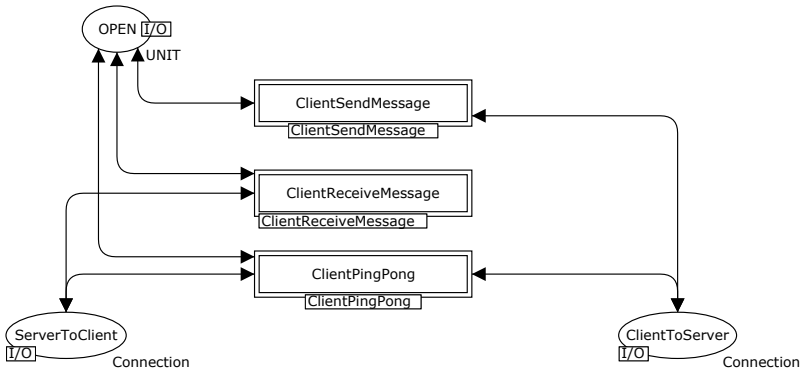


Figure 10. The data transfer phase - client side

is bidirectional, the sending and receiving of data are independent operations. This is reflected by modelling sending and receiving as separate sub-modules of the data transfer module. The sending and receiving of frames are symmetrical for the client and server. Therefore we only discuss the client in the following.

The sending process for a message is shown in Fig. 11. When the client wishes to send a message, this is done by sending a sequence of frames. When the final frame has been sent, the client is *Ready* to send the next message.

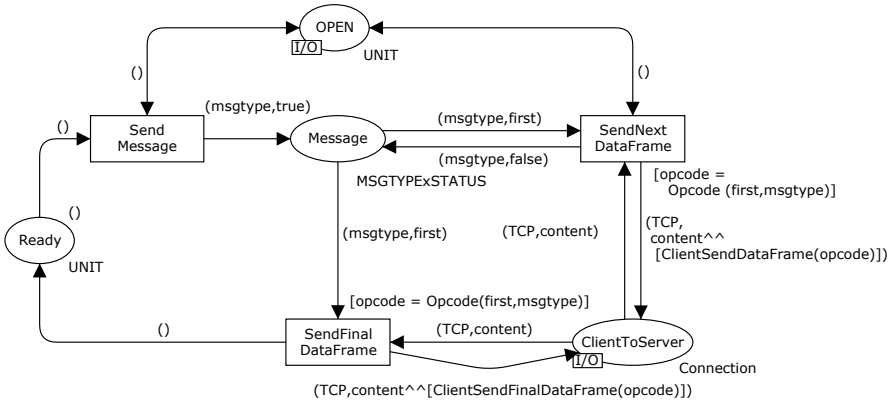


Figure 11. Client side sending of data frames

2.3 Ping and Pong

Ping and pong frames have a rather complex behaviour, as described in the specification document, and is modelled by the module shown in Fig. 12. The complexity lies in the fact that, while a ping frame should be responded to by a pong frame, this may be preempted if the receiver receives another ping frame before responding to the first one. However, if the principal chooses to do so, it can reply to both ping frames. To add to the complexity, pong frames may also be sent unsolicited. The model, however, only specifies that ping and pong frames may be sent and received arbitrarily. This choice has been made to keep the model simple while still covering the behaviour of the protocol.

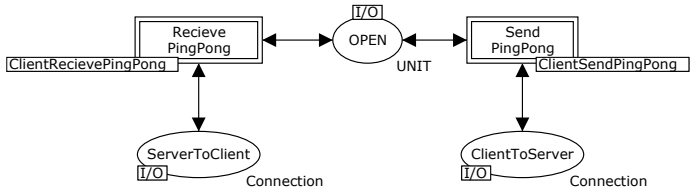


Figure 12. Client side of ping and pong frame processing

Sending ping and pong messages consists of constructing the ping or pong frame and putting it into the TCP channel. Figure 13 shows the sending of ping and pong frames. A function is used on the outgoing arcs of each of the `SendPongFrame` and `SendRecieveFrame` transitions that constructs the respective frames.

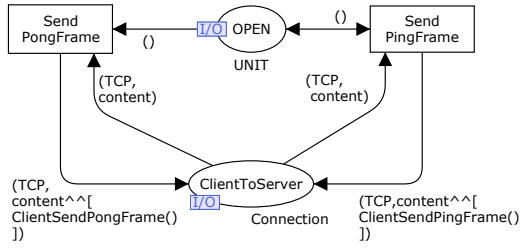


Figure 13. Client side sending of ping and pong frames

The reception of ping and pong frames, shown in Fig. 14, consists of identifying the frame as a ping or a pong frame by its opcode. Upon receiving a ping frame, the principal should send a response as a pong frame as soon as possible

including the same data as the ping frame. Since the protocol model does not contain data and pong frames may be sent unsolicited, the reception of ping frames is not modelled to have any special behaviour.

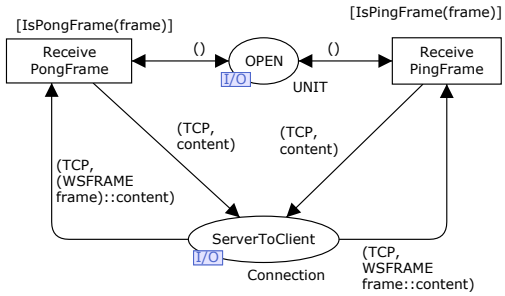


Figure 14. Client side reception of ping and pong frames

2.4 Closing the Websocket Connection

When in the `OPEN` state, either principal may initiate the closing of the WS connection by sending a close frame. The closing process is illustrated in Fig. 15, which shows the client side of the protocol when the server initiates closing. Upon receiving a close frame the client removes the token from the *OPEN* place indicating that the protocol is now in the *CLOSING* state. In the closing state, the client sends a close frame to the server and enters the `CLOSED` state. When both the server and the client are in the *CLOSED* state, the WS protocol is said to be completed and the underlying TCP connection is closed.

3 Verification Model and Initial Results

The purpose of the model presented in Sect. 2 is to serve as a description and executable specification of the WS protocol. This section shows how the descriptive model can be modified to be suited for verification based on explicit state space exploration as supported by CPN Tools. The use of state space exploration caters for model checking of both branching time properties (expressed using, e.g., CTL) and linear-time properties (expressed using, e.g., LTL).

3.1 Finite state space

A first aspect to consider is that the descriptive model of the WS protocol has an infinite state space since there is no bound on the number of frames that can be in transmission over the TCP connection. As an example, a principal

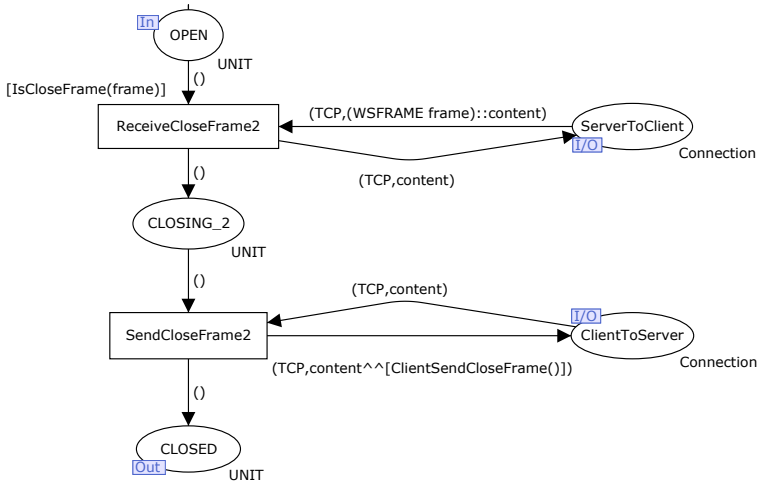


Figure 15. The client side of a server initiated closing handshake

may continuously send a data frame while never reading any data frame. This does not make the descriptive model invalid according to the specification since buffers and congestion control is taken care of by TCP and not in the scope of the WS protocol itself. The descriptive model is therefore modified such that there is an upper bound on the number of frames that can be simultaneously in transmission on a TCP connection. Technically, this bound is enforced by associating a *guard* to each transition that models sending of frames to a TCP connection (e.g., transition *SendPongFrame* in Fig. 13) such that it is required that sending the data does not violate the imposed bound.

3.2 Verification Scope Control

A second element to be incorporated in the verification model is to make it possible to limit the scope of the verification by considering only a subset of the operation of the protocol at a time. This is needed in order to make the verification process incremental, i.e., initiate the verification by considering the smallest possible configuration of the websocket protocol and then gradually include more and more of the functionality until the complete websocket protocol is considered. For this purpose *configuration guards* were added to the model that made it possible in a flexible manner to enable and disable the parts of the model concerned with sending data frames, ping frames and pong frames.

3.3 Abstraction

The third element was to introduce additional abstractions in the model in order to reduce the size of the state space. One example of an abstraction is that the

descriptive models specifies that a message being sent can be of two types: binary or text. This contributes to making the state space larger, but since there are no choices made in the model depending on whether a data frame is carrying binary of textual data it is safe to abstract from this and ignore the type of payload in data frames.

3.4 Verification

An initial verification of the WS protocol concentrated on termination properties of the protocol, i.e., that it is always possible to properly close the WS connection. The WS connection is properly closed when both the client and server is in a closed state and the TCP connection has been closed in both directions. The verification process started by considering only the open and close handshake while disabling the sending of data, ping, and pong frames using the verification scope control described above. In this case the verification model has a single terminal state representing a state where the connection is properly closed. The next configuration considered added the transmission of data. The resulting state space had a number of terminal states some of which represented states where either the server or the client went into the closing handshake *before* a full message currently under transmission had been sent. This highlights an issue where the WS protocol specification is incomplete; a principal needs to clean up the message buffers before entering the closing handshake or that alternatively an additional `TRANSFER` state need to be introduced in the WS protocol to prevent situations where only a partial messages has been sent.

Finally, the sending of ping and pong frames were included in the verification. This highlighted a second issue of unspecified receptions with the protocol in that it is also required for the principals to be able to process ping and pong frames in the closing handshake. This error manifested itself by the presence of terminal states where there were frames in the TCP connection that could not be received. The model was therefore modified such that data, ping, and pong frames can also be received also in the closing handshake. With this modification, it was possible to verify (for the complete verification model) that the WS connection can always be properly closed.

In addition to the two issues related to the design of the WS protocol presented above, the verification process also helped in identifying a number of smaller modelling errors and thereby increasing the confidence in the correctness of the protocol. The state spaces of the WS model configurations considered all had state spaces with less than 3,000 states demonstrating that explicit state space exploration can be a feasible approach also for industrial-sized protocols with the proper abstractions.

4 Implementation Model

Obtaining an implementation of a protocol is another common use of protocol models. A descriptive model can be extended so that implementations can

be automatically generated. This has several advantages. Implementations can be generated for many platforms. Since implementations for different platforms are derived from the same model, all the implementations will have the same behaviour. Thus if the implementation for one platform is correct it gives us confidence that it will be correct for other platforms as well. The implementation is also close to both the descriptive and the verification models. This provides a high degree of assurance that the generated implementations follow the specification and has the properties established using the verification model.

Our approach to adding the information necessary to obtain an implementation is by annotating the CPN model elements in a way that enables a code generator to recognise the annotation and choose an appropriate *code template*. This is the approach taken in [18] where a CPN model is annotated with *pragmatics*. Pragmatics are annotations that assign specific meaning to model elements. Such meaning can be details about the interface, operations that should be performed or denoting the principals of the protocol. The descriptive protocol model together with the pragmatics is then used to generate an implementation of a protocol by associating pragmatics to code templates. The pragmatics do not add any new semantic meaning to the model and are not used in the verification.

In order to obtain an implementation model based on the descriptive model from Sect. 2, annotations needs to be added to several model elements. On the top module of the model, the substitution transitions labelled *Client* and *Server* needs to be annotated to indicate that these substitution transitions represent principals in the protocol. Furthermore, the places *ClientToServer* and *ServerToClient* should be annotated to indicate that these places represent the channels. Figure 16 shows the top module of the implementation model. The *Client* and *Server* are annotated with *<<principal>>* indicating that these substitution transitions are principals in the protocol. Also the places *ClientToServer* and *ServerToClient* are annotated with *<<channel>>* which means that the places represent communication channels in the protocol.

A further refinement is to define the interfaces that other applications use to access the protocol. In the client, the interface includes methods for opening and closing the WS connection as well as sending and receiving messages and ping and pong frames. Information about the interfaces is added to new model elements for each of the methods. Also, the scope for each of the methods should be limited so that, for example, a method for opening the WS protocol returns a

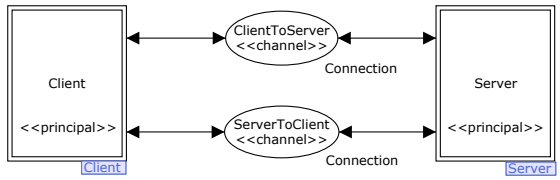


Figure 16. The top level module of the implementation model.

connection handle when WS the handshake is complete, rather than proceeding to send data. This is done by annotating the outgoing arc from the handshake sub-module and adding an annotated element that returns from the method.

Finally, the specifics of the protocol must be annotated with suitable pragmatics. This includes operations that are common to all protocols such as sending and receiving on an underlying channel, and operations that are specific to the considered protocol such as validating the WS keys in the open handshake. Some operations that are not in the descriptive model also need to be added and annotated. An example of this is masking, which is only represented in the model by the *mask* field in the `Frame` colour set. According to the WS protocol specification all data that is sent from the client to the server must be masked. This can be added to the implementation model by a transition in the data sending process annotated with a pragmatic that indicates that the data in the frame should be masked. In the message receiving process on the server, adding support for masking entails adding a transition annotated such that the code generator will recognise that the data should be unmasked.

Many operations are deduced from the structure of the CPN model. Operations like sending and receiving data from the network channels are inferred to exist on transitions that are connected to channel places. For the general flow of the protocol an approach that combines structural analysis and pragmatics is needed. The approach is to identify internal states of the protocol with pragmatics and use these pragmatics together with structural analysis to determine the control flow path of the program. The set of pragmatics is unlikely ever to be complete in the sense that any protocol can be generated using predefined pragmatics. We therefore propose to rely on a package system that provides pragmatics for several domains as well as packages useful in general. In addition, we intend to make it easy to extend the set of pragmatics and adjoining templates for protocol specific operations.

In order to define the possible annotations and their meaning, we rely on an ontology to define the available annotations. The ontology can also be used to restrict the annotated CPN models. For example, one such restriction is that annotations denoting a principal can only be put on a substitution transition, and that this substitution transitions must reside on the top-level module of the model. In order to generate code from an annotated CPN model it is necessary to bind pieces of code to the annotations. This is achieved by using the ontology to relate annotations to code templates. The ontology is currently implemented using a full fledged ontology language, however this approach has shown to be too heavyweight. Hence, further work the ontology describing the pragmatics and their relationships will be created as a simple list of pragmatics with their associated restrictions represented as patterns of the allowed places expressed in a domain specific language.

5 Conclusions and Future Work

In this paper we have introduced a modelling approach for protocol software based on the concepts of descriptive models, verification models, and implementation models. The primary purposes of the descriptive model is to aid in understanding how a protocol works, and to serve as an executable specification based on which models targeting verification and implementation can be derived. We have illustrated our modelling approach by constructing a descriptive model of the IETF Websocket protocol using the CPN modelling language. This model has been used to describe the Websocket protocol and shows, by example, how a descriptive model can be useful for communicating the concepts and operation of a protocol. The model is hierarchical and consists of several modules at various levels of abstraction. By following the suggested modelling approach, it is intended that the descriptive model closely follows the protocol document specification. This in turn gives confidence that the model is describing the protocol. Using a descriptive model that is close to the protocol specification document also provides confidence in models that are based upon the descriptive model and extended for verification and implementation purposes.

The process of getting from a descriptive model to a verification model involves refinements in order to obtain a finite state space, the introduction of additional abstractions, and addition of configuration scope mechanisms in order to control the size of the state space and conduct the verification in an incremental manner. For the web socket protocol, we demonstrated that this resulted in small state spaces. Our initial verification of the web socket protocol additionally identified omissions in the protocol specification related to the close of connections during message transfer and unspecified receptions of data, ping, and pong frames during the closing handshake. With proper modifications to the verification model, we were able to verify the protocol ensures correct termination of connections.

The process of getting from the descriptive model to the implementation models involves annotations using pragmatics following the approach of [18]. The pragmatic are used to specify, e.g., principals and interfaces, and is used to map the model elements to code templates and the underlying execution platform. The implementation model presented in this paper is of a form where it can be used as a basis for code generation following the approach presented earlier in [18].

Maintaining consistency between the descriptive, the verification and the implementation models is an important issue in our approach. One way to do this is to use tool support to force the models to evolve in parallel. In this scenario, structural changes can only be done on the descriptive specification model, while arc inscriptions can be changed in the verification and implementation models. This way the specialised models could add annotations and details needed to limit the state space and make code generation possible while still being structurally identical to each other and the descriptive specification model. A drawback with this approach is that conflicts concerning the level of detail in the model structure may arise. Another approach is to allow differences between the

models, but make it part of the methodology that the differences should only include simple additions in the implementation model. These additions should only be added when it is absolutely necessary. Also the effects of such additions should be thoroughly investigated. A drawback of this method is that it requires a large amount of modelling discipline. A hybrid between the two approaches where tools give stern warnings whenever structural equivalence is violated is also an option. This way the differences between the models will amount to little more than different views on the descriptive model preserving its semantics while still being flexible enough to accommodate necessary modifications for verification and implementation. For the example of the WS protocol, the only modifications that are needed for code generation are the additions of pragmatics that do not change the semantics of the model. And the only modifications for verification is instrumentation that allows for limiting the scope of the verification and limiting the number of packets that is sent.

As part of future work, we also plan to use the verification model presented here to verify additional connection management properties of the Websocket protocol. Furthermore, we plan to add proxies to the model so we can verify the Websocket protocol with proxy servers which is an area where the operation of the Websocket protocol is highly complex and where we expect to be able to identify number of non-trivial design issues. In parallel with verification, we also plan to use the implementation model derived from the descriptive model to generate an implementation of the Websocket protocol using the approach in [18]. Here, we plan to validate the derived implementation using test-suites [19] that are created to validate implementations of the Websocket protocol.

References

1. ISO/IEC 15437. Information technology. Enhancements to LOTOS (E-LOTOS), September 2001.
2. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 49–70, 2004.
3. M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proc. of SIGCHI Conference on Human Factors in Computing Systems*, pages 557–566, 2007.
4. C. Choppy, L. Petrucci, and G. Reggio. A Modelling Approach with Coloured Petri Nets. In *Proc. 13th International Conference on Reliable Software Technologies*, volume 5026 of *LNCS*, pages 73–86, 2008.
5. B. Genest, A. Muscholl, and D. Peled. Message sequence charts. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 537–558. Springer, 2004.
6. S. Gordon and J. Billington. Analysing the WAP Class 2 Wireless Transaction Protocol Using Coloured Petri Nets. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 207–226. Springer, 2000.
7. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

8. Internet Engineering Task Force. *The WebSocket protocol*, December 2011. `http://tools.ietf.org/html/rfc6455`.
9. ISO9074. Information Processing Systems - Open Systems Interconnection: ESTELLE (FOrmal Description Technique Based on an Extended State Transition Model.
10. ITU-T. Z.120: Message Sequence Charts (MSC), 1996.
11. ITU-T. Z.100-Z.106: Specification and Description Language (SDL), 2010.
12. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
13. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.
14. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Prot ocol for Mobile Ad-hoc Networks. In *Proc. of Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer, 2004.
15. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of Formal Methods for Industrial Critical Systems*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
16. C. Ouyang and J. Billington. Formal Analysis of the Internet Open Trading Protocol. In *Proc. of Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops*, volume 3236 of *LNCS*, pages 1–15. Springer, 2004.
17. K. Simonsen and L.M. Kristensen. *Descriptive specification model of the Websocket protocol*. `http://gs.hib.no/ws-model/WSProtocol.cpn`.
18. K. I.F. Simonsen. On the use of Pragmatics for Model-based Development of Protocol Software. In *International Workshop on Petri Nets and Software Engineering*, 2011.
19. Tavendo GmbH. *Project Web Site*. `http://www.tavendo.de/autobahn/testsuite.html`.
20. CPN Tools. *Project Web Site*. http://www.cpntools.org/.

Chapter 9

Generating Protocol Software from CPN models Annotated with Pragmatics

Generating Protocol Software from CPN models Annotated with Pragmatics

Kent Inge Fagerland Simonsen[1,2], Lars M. Kristensen[1], and Ekkart Kindler[2]

[1] Department of Computer Engineering, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
[2] DTU Compute, Technical University of Denmark, Denmark
Email: {kisi,ekki}@dtu.dk

Abstract. Model-driven software engineering (MDSE) provides a foundation for automatically generating software based on models that focus on the problem domain while abstracting from the details of underlying implementation platforms. Coloured Petri Nets (CPNs) have been widely used to formally model and verify protocol software, but limited work exists on using CPN models of protocols as a basis for automated code generation. The contribution of this paper is a method for generating protocol software from a class of CPN models annotated with code generation pragmatics. Our code generation method consists of three main steps: automatically adding so-called derived pragmatics to the CPN model, computing an abstract template tree, which associates pragmatics with code templates, and applying the templates to generate code which can then be compiled. We illustrate our method using a unidirectional data framing protocol.

1 Introduction

Model-driven software engineering (MDSE) [4] provides a foundation for highly automated generation of software based on models. The use of models allows software designers to focus on the problem domain and abstract from the details of underlying implementation platforms. If the MDSE process uses modelling languages with a formal semantics, we gain the additional advantage that the models can be verified, e. g. by model checking [2]. The combination of formally verified models from which code is generated automatically increases the confidence in the resulting implementation being correct with respect to the formally specified properties.

Coloured Petri Nets (CPNs) [9, 10] have been widely used for formal modelling and verification of protocol designs [14, 3], but limited work has been done on developing methods that support the use of CPN models as a basis for automated code generation of protocol software [13, 16]. CPNs extend ordinary Petri nets with a programming language for defining data types and using inscriptions for modelling data and data manipulation. In addition, CPNs provide a module concept that allows large CPN models to be structured as a hierarchically related set of modules. CPN uses Standard ML (SML) as programming language.

The contribution of this paper is a method for automated code generation from CPN models based on a modelling methodology for constructing *descriptive models* of protocols and on adding *code generation pragmatics* to the CPN models. The notion of descriptive models is firstly intended as a means for creating models that are helpful in understanding and conveying the operation of the considered protocol. Secondly, a descriptive model is close to a verifiable version of the same model and sufficiently detailed to serve as a basis for automated code generation when annotated with code generation pragmatics. The relationship between descriptive models and verification models was discussed in [12]. In this paper, we concentrate on the pragmatics, the modelling methodology for constructing descriptive models, and on the steps of the code generation.

The pragmatics that we integrate into the CPN language are syntactical annotations that are associated with CPN model elements. The primary purpose of the pragmatics is to add enough details for generating code without cluttering the model and making it verbose which would ultimately render it unreadable and too complex for verification purposes. It should be noted that pragmatics are purely syntactical annotations for code generation purposes, and hence our method does not affect the formal semantics of CPNs. The pragmatics fall into three types: structural, control flow, and operation pragmatics. Our method defines a set of core pragmatics that are applicable to all protocols. In addition, our method is extensible in that it allows the modeller to easily add new pragmatics if required by a specific protocol or a specific protocol domain under consideration.

The code generation consists of three main steps, starting from a CPN model that the modeller has annotated with a set of pragmatics that makes the protocol structure and the control flow explicit. The first step is to automatically compute for the CPN model, a set of *derived pragmatics* that identify common control flow structures and operations, such as sending and receiving packets, or manipulating states. In the second step, an *abstract template tree* (ATT) is constructed providing an association between pragmatics and code generation templates. Essentially, every node of the ATT will be associated with a code template. In the third step, the ATT is traversed and code is emitted by invoking the code templates associated with each node of the ATT rather than translating SML. A key feature of our method is that the generated code resembles what a human programmer would have developed. This is advantageous with respect to code inspection, maintainability, and performance.

This paper is organised as follows. Section 2 presents our modelling methodology and the explicit pragmatics. In Sect. 3, we introduce automatically derived control flow and operation pragmatics. In Sect. 4, we cover ATTs and their use in code generation. In Sect. 5, we discuss related work, and, in Sect. 6, we sum up conclusions and outline directions for future work. Due to space limitations, we cannot present our method in full detail here. These can be found in the technical report [22]. A very early and preliminary version of these ideas was presented as an extended abstract [21]. A prototype of a tool supporting the approach presented in this paper is available: PetriCode. For more information

on the tool, we refer to the tool's home page [18]. We assume that the reader is familiar with the basic concepts of Petri nets (places, transitions, tokens, enabling, and firing rule), and we introduce CPN specific concepts only briefly as we proceed. A comprehensive introduction to CPNs is given in a textbook [10].

2 Modelling Methodology and Explicit Pragmatics

To present our modelling methodology we use, as a running example, a unidirectional framing protocol. The overall service provided by this protocol is to send *messages* of arbitrary length from a sender to a receiver by splitting up the message into smaller *packets* sent across a unidirectional channel. The channel is assumed to be reliable and to preserve the order of the transmitted packets. The protocol uses a *final bit* in each transmitted packet indicating whether the payload of the packet is the final (last) part of the larger message. As we proceed with presenting the CPN model, we introduce the basic set of explicit pragmatics that are central to our method and which the modeller uses as part of the construction of the CPN model. Pragmatics are by convention written in ⟨⟨ ⟩⟩ to distinguish them from, e.g., place and transition names and SML inscriptions.

2.1 Protocol System Level

Figure 1 shows the top-level module of the CPN model which constitutes the *protocol system level*. The purpose of the protocol system level is to specify the *protocol principals* and the *channels* connecting them. This module has three CPN *substitution transitions* (transitions with double lined borders) named Sender, Channel, and Receiver. Substitution transitions constitute the basic structuring mechanism of CPNs and each substitution transition has an associated submodule modelling the details of the compound behaviour represented by the substitution transition. The two substitution transitions Sender and Receiver represent the two principals of the protocol, and the substitution transition Channel represents a channel between them. We use the ⟨⟨principal⟩⟩ pragmatic to specify which substitution transitions represent protocol principals, and the ⟨⟨channel⟩⟩ pragmatic to specify substitution transitions representing channels. The channel pragmatic has three associated *properties* specifying that the channel is unidirectional, reliable (i. e., the channel does not loose packets), and that it preserves the order of packets. Our modelling methodology includes a set of channel modules for common channel types and the specific module to be used in the model is selected based on the properties specified for the channel pragmatic. The two *socket places* (places connected to a substitution transition) SenderChannel and ReceiverChannel connecting the principal substitution transition to the Channel are implicitly considered *channel places* which means that messages (tokens) added and removed from these places are considered to be sent and received, respectively. In CPNs, a socket place can be associated with a *port place* in the submodule of the substitution transition. This has the effect that the two

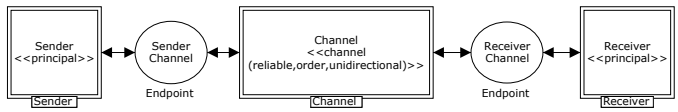


Fig. 1. The protocol system level

places, conceptually, become the same place; this way, sockets provide the means by which modules in CPNs exchange tokens.

We require in our modelling methodology that the protocol system module consists of one or more substitution transitions representing principals. A socket place at the protocol system level can be connected to at most one principal substitution transition and at most one channel substitution transition. This requirement is needed since we use the socket places connecting principals and channels to identify which channel or principal a message is intended for.

The concept of a channel represents a means for communication between *endpoints* as determined by the colour set (data type) `Endpoint` which consists of a name identifying the endpoint and an input and an output buffer for packets transmitted on the channel. In CPNs, the data type of a place is by convention written below the place and determines the kind of tokens that may reside on the place. The protocol system level and the modelling of channels are parameterised by colour sets (data types) used to identify channels and the specific packets transmitted. This means that we assume only the existence of these two types and do not make any assumptions on how they are realised. The concrete implementation of the `Packet` colour set in a protocol model depends on the protocol data units exchanged among the principals in the protocol under consideration. For code generation purposes, the implementation of the `EndpointId` colour set depends on the concrete channel used to realise the communication between the principals. If for instance, the channel is realised using the transport layer of the TCP/IP protocol stack, then the Endpoint colour set will consist of a host (IP address) and a port (a process). Hence, in a TCP/IP context, an endpoint can be implemented as a TCP/IP socket. The colour sets also have an associated class of functions that play a central role in being able to recognize common structural patterns in the CPN models, which are captured by the operation pragmatics to be presented in Sect. 3.

2.2 Principal Level

The submodules of principal substitution transitions in the protocol system module constitute the *principal level modules*. Each principal level module specifies the *services* that are provided by the corresponding principal and the life-cycle of the principal. In addition to specifying constraints on the order of service uses, the principal level modules may also model the state to be maintained across invocation of the services. The explicit modelling of the methods that constitute the service is required in our method in order to generate code that can be integrated into different code contexts.

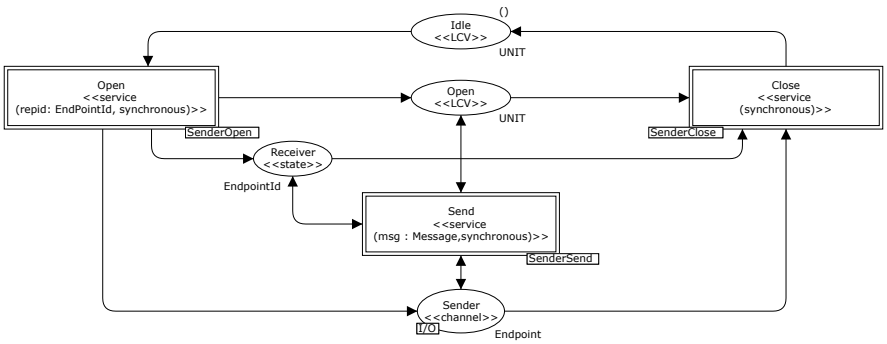


Fig. 2. The Sender module

Here, we concentrate on the sender principal as a representative example. Figure 2 shows the principal level CPN module for the sender. This module is the submodule of the Sender substitution transition in Fig. 1. The module has three substitution transitions annotated with the ⟨⟨service⟩⟩ pragmatic to indicate that they represent services that are to be exposed by the implementation, i. e., be externally visible. In this case, the sender has three services: Open (for opening the communication with the receiver), Send (for sending a message), and Close (for closing the communication with the receiver). The parameters of the ⟨⟨service⟩⟩ pragmatic specify the parameter and return types, and properties of the services. In this case, all three services provided by the sender principal are synchronous services as specified by the synchronous property of the ⟨⟨service⟩⟩ pragmatics. Our method also supports asynchronous services which, however, are not discussed here (see [22] for details).

The principal can be in two different states as modelled by the places Idle and Open with the colour set `UNIT` containing just a single value `()` (called unit and representing a black token). When there is a unit token on Idle, this means that no communication is initialised, and when there is a unit token on Open this means that messages can be transmitted to the receiver. A third implicit state is also possible when neither the Idle nor Open places have a token. This state is reached when the client is busy opening, sending or closing. A place modelling a principal life-cycle state is annotated with the ⟨⟨LCV⟩⟩ pragmatic (Life Cycle Variable). The open service can be invoked only when the principal is in Idle and, once Open, messages can be sent, and the communication can be closed. In the latter case, the sender returns to the Idle state. The sender maintains another state variable Receiver, which represents the endpoint created by Open, and is used by Send in order to send messages. State variables are indicated using the ⟨⟨state⟩⟩ pragmatic. The *port place* Sender (bottom) is associated with the SenderChannel socket place in Fig. 1 and hence any token added (removed) to Sender will be added (removed) to SenderChannel and vice versa. In the sender module, the place Sender has been annotated with the ⟨⟨channel⟩⟩ pragmatic

which is derived from the fact that the associated socket place at the protocol system level is connected to a channel substitution transition (see Fig. 1).

The principal level modules do not specify how a wrong use of the services should be handled, e. g. when the send service is invoked in a state where the sender is not Open. The associated error handling is platform dependent.

2.3 Service Level

The submodules of the substitution transitions annotated with ⟨⟨service⟩⟩ on the principal level specify the detailed behaviour of the principals for each of the principal's services. The detailed behaviour is modelled in a control flow oriented manner using ⟨⟨ID⟩⟩ pragmatics on places to make the control flow explicit. Modelling the services in a control flow oriented manner serves two main purposes. The first purpose is to provide for comprehensible models in that the explicit control flow provides a reading path to the model of the service. This is in contrast to a pure event-oriented approach to modelling (as discussed in [3] for example) from which no control flow is explicit and which consists of modelling a protocol principal using a single place to represent its state and a set of transitions connected to this place which changes the state of the principal depending on packets sent and received. The second purpose of modelling in a control flow oriented manner is to automatically generate code with a structure that resembles what a human programmer would implement. This makes it easier to inspect and maintain automatically generated code, and provides code with better performance since it reflects the intended use of the constructs provided by the target programming language.

As a representative example of a service level module, we consider the send service of the sender principal which is shown in Fig. 3 (left). At this level, the ⟨⟨service⟩⟩ pragmatic is used on ordinary (non-substitution) transitions to indicate the single entry point for the corresponding service primitive. Hence, it is possible to have only one transition annotated with ⟨⟨service⟩⟩. The message to be sent is represented by the parameter `msg` of the ⟨⟨service⟩⟩ pragmatic. Transitions representing the termination/completion of the service are annotated with the ⟨⟨return⟩⟩ pragmatic. We assume that there is exactly one transition in a service level module that is annotated with ⟨⟨return⟩⟩. In general, the ⟨⟨return⟩⟩ pragmatic may take parameters representing return values. The parameters for the open service specifies the endpoint of the receiver principal. These parameters are stored in the Receiver state variable and also an endpoint is created on the Sender channel place which the sender will use for sending packets.

Places modelling the control flow in the send primitive are annotated with an ⟨⟨ID⟩⟩ pragmatic. From a control flow perspective, the send operation has an overall sequence (starting at transition Send and ending at transition Completed), and a repeat-until loop (starting at place Start and ending in place PacketSent). The operation of the send primitive is to first partition the message to be sent into a sequence of smaller sub-messages which is placed on Outgoing. In CPNs, the expression associated with arcs specifies the tokens to be removed and added when transitions occur. The expressions may contain free variables which

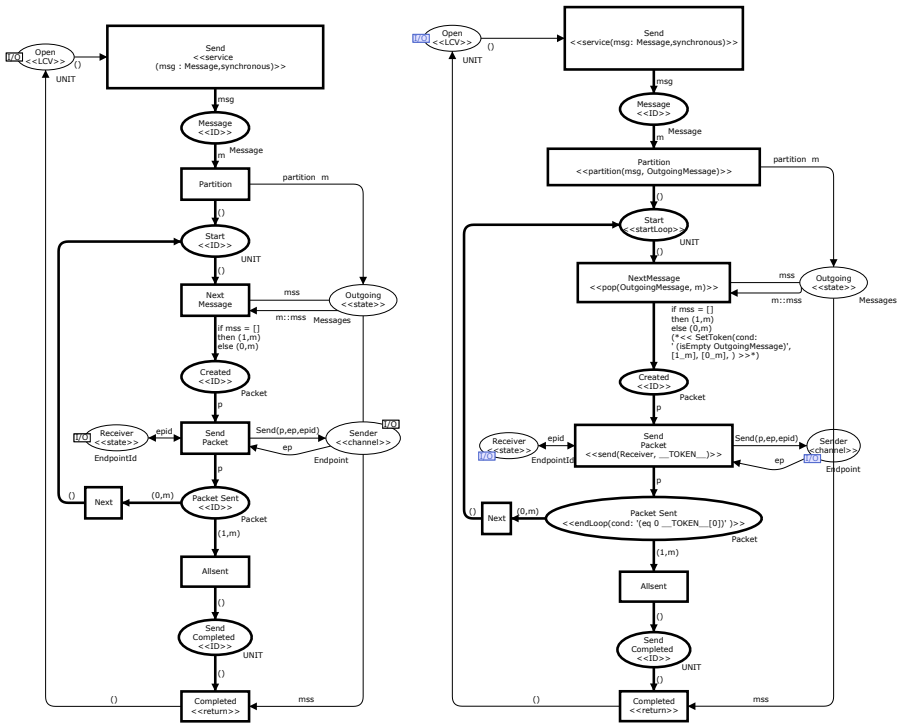


Fig. 3. The SenderSend module with explicit pragmatics (left) and derived pragmatics added (right). The derived pragmatics are discussed in Sect. 3

determines possible modes in which a transition may occur. As an example, the Partition transition in Fig. 3 (top) has a variable `m` of type `Message` which in an occurrence of Partition will be bound to the value of a token present on place Message. When the transition occurs, it will remove the corresponding token from Message, and add tokens to the outgoing places Start and Outgoing obtained by evaluating the expressions on the corresponding arcs. The `partition` function takes a message as argument, and constructs a list of submessages that is added as a token on place Outgoing. Also, a unit token will be added to place Start. The sender then executes a loop in which a packet is sent for each sub-message.

The modelling of the sender includes some intermediate states (e. g., SendCompleted) which makes the model more verbose, but is used in our method for recognising control flow constructs. It is worth noting that, in the model of the send service, the token is removed from Open while the send operation is in progress; this prevents any further sending or invocation of close while a send operation is executed (the protocol is not designed for concurrent sends).

3 Derived Code Generation Pragmatics

Before we discuss how the actual code generation works, we discuss some additional pragmatics which are used by the code generator. Since these pragmatics can be automatically derived from the net (model) structure and the arc inscriptions, these pragmatics are called *derived pragmatics*.

The first kind of pragmatics concerns the control flow, which indicate how the net structure of a service module is decomposed into control flow blocks that constitute the ATT (see Sect. 4 for more details). Therefore, this kind is called *control flow* pragmatics. The second kind of pragmatics, called *operation pragmatics*, helps generating the code for the actual operations that are to be executed. We explain these pragmatics by the help of the SenderSend module example, which was shown in Fig. 3.

3.1 Block Structure and Control Flow Constructs

CPNs (and Petri nets in general) do not enforce any particular structure with respect to the modelling of the control flow of the service primitives. In order to be able to generate code that uses the control flow constructs of typical programming languages, we assume that the net structure induced by places of the service level modules that are marked with ⟨⟨ID⟩⟩ can be decomposed into *control flow blocks*. For the Send primitive in Fig. 3(left), the part corresponding to control flow blocks has been graphically indicated in bold. Formally, the block structure decomposition is defined by having different types of blocks, which inductively define the block structure of a net. Due to space limitations, we cannot go into the details of this definition here (see [22] for the technical and formal details). There are four types of blocks: *atomic*, *choice*, *loop*, and *sequence*, and two of these patterns are sketched in Fig. 4. The pattern in Fig. 4(left) captures that an atomic block consists of a start place (top), a single transition, and an end place (bottom). The pattern in Fig. 4(right) specifies that a loop block has a single start place (top), a body (indicated by ...) and an end place (bottom) and a single transition (right) capturing the iteration by connecting the end place and the start place. For the SenderSend module in Fig. 3 the control flow can be decomposed into a block, which is a sequence, where the first element of that sequence is an atomic block, the second is a loop, which again consists of a sequence of two atomic blocks.

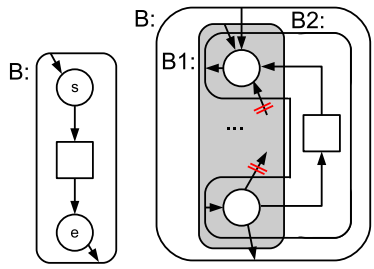


Fig. 4. Blocks: Atomic and loop

For code generation purposes we, systematically decompose each service level module into blocks where the containment of the blocks defines the structure of the ATT. For the actual code generation, it is sufficient to identify the start and end of loops and choices – actually the places where they start and end – with some additional pragmatics: ⟨⟨startLoop⟩⟩, ⟨⟨endLoop⟩⟩, ⟨⟨branch⟩⟩, and

⟨⟨merge⟩⟩. For the SenderSend module, these additional pragmatics are shown on the right-hand side of Fig. 3. Note that the ⟨⟨endLoop⟩⟩ has a parameter, which represents the exit *condition* of the loop. The ⟨⟨branch⟩⟩ pragmatic for a choice has a *condition* parameter too, but we do not have a choice in this example. Our technology comes with a simple syntax for formulating these conditions, which resembles the syntax of Lisp. In our example, the expression `(eq 1 __TOKEN__[0])` checks whether the first component (referred to by index 0) of the control flow token (referred to by `__TOKEN__`) is equal to 1, which reflects the inscription of the arc leaving the loop. We use this condition parameter and the specific syntax for conditions in order to be independent of SML. By adding the condition as a parameter of the ⟨⟨endLoop⟩⟩ pragmatic, we do not need to restrict the annotations of CPN models – at the price of, sometimes, being forced to add the condition of the ⟨⟨branch⟩⟩ and ⟨⟨endLoop⟩⟩ pragmatics manually.

3.2 Operation Pragmatics

The *operation pragmatic* is associated with transitions and describes an operation associated with the execution of the transition in a programming language independent way.

The right-hand side of Fig. 3 shows three examples of these pragmatics. The ⟨⟨send⟩⟩ pragmatic is an example of a protocol independent pragmatic. It represents sending a message to another principal, which is represented by the pattern for this transition. The parameters of the ⟨⟨send⟩⟩ pragmatic define the target of the send (here identified by the end point on place Receiver) and the actual message to be sent (here, the message is contained in the current token).

The other two operation pragmatics are more specific to this particular protocol: ⟨⟨partition⟩⟩ splits a message into the sequence of chunks that are supposed to be sent – actually a list of these chunks. The ⟨⟨pop⟩⟩ operation, obtains and removes one chunk from the list.

As mentioned above, some of the operation pragmatics are part of the general method, and for these there will be direct code generation support available defined by so-called template bindings. These bindings are discussed in Sect. 4. In addition, a protocol developer can add own protocol specific pragmatics; in that case, the developer must provide the corresponding templates and bindings at some point in order to generate the code.

4 Abstract Template Trees and Code Generation

The actual generation of code from a CPN model annotated with explicit and derived pragmatics proceeds in three phases: The first phase is the construction of an ATT which serves as an intermediate representation in the code generation. The second phase binds code generation templates to the nodes of the ATT corresponding to the target platform under consideration. The third phase is to traverse the ATT and invoke the code generation templates in order to emit

code. Below, we illustrate the three code generation phases using the annotated send service module shown in Fig. 3 (right) as an example. The target platform considered in our example is the Groovy programming language. Groovy is a multi-paradigm language that runs on the Java Virtual Machine. It was chosen as a target because it is an optionally typed multi-paradigm language with features that makes it fairly easy to generate code for while still being a realistic platform for industrial applications.

An ATT is an ordered tree of nodes and resembles abstract syntax trees. The two major types of nodes in the ATT are *leaf* (operation) nodes and *container* nodes. A leaf node does not have children and contains pragmatics for one or more sequential operations such as sending on a channel or accessing a state variable. A container node has in addition to associated pragmatics, an ordered list of child nodes. The types of container nodes at the service level corresponds to the different types of blocks introduced in Sect. 3. The root node of the ATT represents the entire protocol system. The generation of the ATT is implemented by a guided walk through the CPN model. This walk starts at the protocol system module and, for each ⟨⟨principal⟩⟩ pragmatic, it generates a corresponding node in the ATT. On the next level, the generator looks for modules annotated with a ⟨⟨service⟩⟩ pragmatic and adds corresponding nodes. Each service module contains exactly one transition with the ⟨⟨service⟩⟩ pragmatic, which is the starting point for the method modelled by the sub-module. The subsequent set of nodes is constructed according to the block structure rules described in Sect. 3.

The sub-ATT corresponding to the sender send service is shown in Fig. 5. The node at the top represents the sender send service. The child nodes of the Send node correspond to the overall sequence performed by the send service: partitioning the message, executing the loop where submessages are sent, and then completing the service. The child nodes of the Start node correspond to the body of the loop.

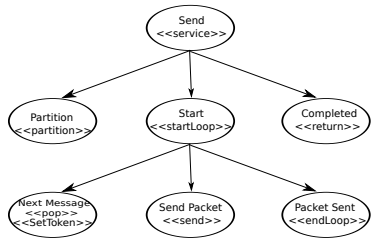


Fig. 5. Sub-ATT for sender send service

When the ATT has been generated, in order to generate code for a particular platform, the pragmatics represented by the nodes of the ATT must be bound to code generation templates. This is done by means of a *template descriptor*. A template descriptor contains a line for each pragmatic that need to be translated into code for a specific platform. The template binding for a pragmatic contained in an ATT node is determined by the line for the pragmatic contained in the template descriptor.

The template descriptor is specified in a simple domain specific language (DSL). An extract of the binding descriptor for generating Groovy code covering three of the pragmatics from Fig. 5 can be seen in Listing 1. Each line of the template descriptor consists of a name followed by a left-parenthesis followed by key value pairs where the keys can be `pragmatic` which contains the name of

the pragmatic, `template` which corresponding value is the path to the template, `isContainer` which indicates whether this pragmatic denotes a container or a `isMultiContainer`. The multi-container flag is primarily an implementation detail in our tool used to indicate whether or not the container is of type loop or choice.

Listing 1. Extract of binding descriptor for the Groovy platform

```
partition(pragmatic: 'partition', template: 'groovy/partition.tmpl',
          isContainer: false, isMultiContainer: false)
send(pragmatic: 'send', template: 'groovy/send.tmpl',
          isContainer: false, isMultiContainer: false)
startLoop(pragmatic: 'startLoop', template: 'groovy/loop.tmpl',
          isContainer: true, isMultiContainer: true)
```

Generating the protocol software consists of traversing the ATT and invoking the associated templates for each node as described by the template binding. When a pragmatic is transformed to code, its template is run through the template engine together with a number of parameters given by the pragmatic definition and the CPN structure. The templates are sown together by replacing a special tag in the container templates, `%%yield%%`, with the text of the underlying templates in order.

As an example of a container template, the template for the loop pragmatic for the Groovy language is given in Listing 2 (left). The template creates a while-loop which continues while the `__LOOP_VAR__` variable is true. The body of the loop is populated by replacing the `%%yield%%` directive with the code generated by the templates of the sub-nodes in the ATT. The `__LOOP_VAR__` is updated at the end of the loop by the ⟨⟨endLoop⟩⟩ pragmatic which is always present as the last child element of a loop. The ⟨⟨send⟩⟩ is an example of an operation pragmatic. Listing 2 (right) shows the template for the ⟨⟨send⟩⟩ pragmatic which requires two parameters: one is the name of the socket that the message should be sent on, and the other is the variable that holds the message to be sent.

Listing 2. Examples of templates for loops (left) and send (right)

```
%%VARS:__LOOP_VAR__%%
__LOOP_VAR__ = true
while(__LOOP_VAR__){
    %%yield%% }
```

```
${params[0]}.getOutputStream()
    .newObjectOutputStream()
    .writeObject(${params[1]})
%%VARS:${params[1]}%%
```

As an example of the generated code, the loop in the sender service in the Sender principal is shown in Listing 3. The loop is started by defining a variable, `__LOOP_VAR__`. After the `__LOOP_VAR__` is defined, the loop is entered. Inside the loop, the next fragment is code from the template bound to the ⟨⟨pop⟩⟩ pragmatic. This code removes the first element from `OutgoingMessage` and assigns it to variable `m`. Then, the code for the ⟨⟨setToken⟩⟩ pragmatic on the arc between the transition NextMessage and the place Created is generated. This code sets the `__TOKEN__` variable in the code according to the conditional statement in the pragmatic: if `OutgoingMessage` is empty then the message is prefixed by 1, otherwise it is prefixed by zero. The next pragmatic that is found on the

control flow path is the ⟨⟨send⟩⟩ pragmatic on the transition Send Packet. The socket `Receiver` is used to send the value of the `__TOKEN__` variable. Finally, the template associated with the ⟨⟨endLoop⟩⟩ pragmatics has generated the code for updating `__LOOP_VAR__` according to the conditional expression given as a parameter to the ⟨⟨endLoop⟩⟩ pragmatic.

Listing 3. The generated code for the loop of the sender send service

```
__LOOP_VAR__ = true
while(__LOOP_VAR__){
def m = OutgoingMessage.remove(0)
if(OutgoingMessage.size() == 0){
__TOKEN__ = [1,m]
} else {
__TOKEN__ = [0,m]
}
Receiver.getOutputStream().newObjectOutputStream().
          writeObject(__TOKEN__)
__TOKEN__
__LOOP_VAR__ = 1 == __TOKEN__[0]
}
```

5 Related Work

The goal of our code generation method is to generate code from models close to descriptive models that are amenable to verification with little or no modification. Also, the code that is generated should be readable, portable and maintainable. Furthermore, we would like to be able to easily integrate our code into third party software and have a great deal of flexibility in the way code is produced.

There are many methods for modelling and analysing protocol software using languages such as High Level Petri Nets [7], temporal Petri Nets [23], ESTELLE [5] and LOTOS [15]. Some methods support automatic code generation such as state charts [24], SPI [20], SDL [8] and UML [1]. Due to space limitations, we focus our discussion on approaches that use general purpose languages (UML and CPNs) equipped with additional information for a specific domain. In the rest of this section, we discuss several related works and finally, at the end, contrast and sum up the key differences between each of the related work items and our approach.

In [19], possible methods for code generation from high level Petri Nets (HLPNs), such as CPNs, are discussed and a new hybrid of the discussed approaches is presented. The general methods for code generation from HLPNs are, according to [19]: structural analysis, simulation based, and reachability graph based. The method proposed in the paper is a hybrid of simulation based and structural analysis methods.

In [16], the author describes an approach for generating code from CPNs for an access control system. The generation takes advantage of the fact that CPNs use the SML programming language for all inscriptions. This means that it is

fairly simple to generate SML code that simulates the CPN in SML code. And by using external libraries, the CPN can interact with other devices through a specialized protocol for access control systems. The paper also presents a case study where the techniques discussed are used to generate an access control system for an industrial actor. A somewhat similar approach is also taken in [11] where the core of a tool for scheduling courses of actions is created based on a CPN model. The model is extracted from the modelling tool and executed as an SML program.

Process-Partitioned CPNs (PP-CPNs) [13] have been used to automatically generate code for several purposes including protocols. Code is generated from PP-CPNs by first translating the PP-CPN into a control flow graph (CFG), then translating the CFG into an abstract syntax tree (ASTs), first of an intermediary language then to an AST that is dependent on the target platform. From the platform dependent AST, code is generated. In [13], PP-CPNs are used to model and obtain an implementation for the DYMO routing protocol using the Erlang programming language and platform.

In [17], a UML profile named Graphical Protocol Description Language (GPDL) is used together with a textual language called GAEL to model and generate code for protocols. The approach uses stereotypes to annotate UML diagrams with information used for code generation. The stereotypes and GAEL annotations are used though a series of transformations to generate code. In [17], the authors produce SDL code, but are able to produce code for any platform.

In the terminology of [19], our code generation method is based on structural analysis, but it is also based on user input in the form of explicit pragmatics. The pragmatics coupled with templates makes it possible to be platform independent and create readable and maintainable code which has an interface based on the services described on the principal level. The template approach also gives the modellers flexibility, by modifying the templates, to create code in their own style. The methods presented in [19, 16, 11] are all based on simulating the models. The simulation methods conflicts with our goals of readable code as the purpose of the code can easily be lost in the details of the operations of the simulator. Also the code generated by the simulation methods is not likely to be efficient in particular due to the complex enabling computation that needs to be performed in each step of the execution. The method presented in [13] constrains the models more than our method since we have the possibility to add more pragmatics to expand the range of functionality. Also, [13] does not model how services can be used, so it does not allow the modeller to control how third party applications could be integrated with the generated code. In contrast to our approach, the approach in [13] is also bound to the Erlang platform where our approach, through templates is platform independent. Also, our approach provides more flexibility in the operations that can be modelled by allowing users to define additional pragmatics. The approach in [17], despite being based on UML, has several similarities with our approach such as annotating the models with stereotypes which are similar to our pragmatics. However, the stereotypes are predefined in a UML profile and does not offer the same flexibility in mod-

elling as our templates that may be user defined. Also the GPDL models use a separate language, GAEL, to provide additional information in addition to the platform information which in our case is contained in templates and template bindings.

6 Conclusions and Future Work

In this paper, we have presented a method for automatically generating code for protocol software from CPN models. The method was discussed by a simple, but complete example of a communication protocol. The code generation approach has been realized in a tool that was used to generate the code examples in this paper. The tool can be accessed from the project website [18].

The main objective of our method is that code can be generated from what we call *descriptive models*. Descriptive models are typically used for understanding and explaining how a protocol works on a high level of abstraction. Descriptive models focus on concepts and not on technical details and, in many cases, these models can be used – with some tweaking – also for analysing and verifying protocols. Today, it is typical practise to use models for analysing a protocol and its specification and for verification of the protocol. Then, the protocol software is implemented manually based on these models. Our method makes it possible to use the same descriptive model for analysis and verification as well as for code generation – in both cases, the models are moderately extended.

In our method, we chose to use Coloured Petri Nets (CPNs) [9] as modelling language for descriptive models since they have successfully been used for modelling, analysing, and verifying various kinds of systems [10] for a long time now. Over the time, specific modelling styles, principles, and disciplines have developed for using CPN for that purpose. These styles and principles are mostly used informally – sometimes not even mentioned at all. In our method, we needed to make them into more rigorous rules.

Since descriptive models are conceptual in nature and on a high level of abstraction, they often do not capture some technical aspects and implementation details. Examples of such information not contained in descriptive models are the API and the interface for calling the services or operations of a protocol. Our method caters for that by *pragmatics* that can be added to different elements of the model. This way, it is possible to attach additional information without compromising the overall structure of the original model. And our example shows, that all relevant technical information can easily be added to the model in this way. We argue that adding pragmatics will not add significantly to the modelling effort. One reason for this is that explicit pragmatics, to a large extent, represent concepts the modeller would be aware of while modelling, so adding them should add little more time than looking up and adding the pragmatics. Also, derived pragmatics are added automatically and therefore require no additional action from the modeller. Adding new pragmatics is relatively simple since all that is required is to add templates and describe the pragmatic and template bindings in simple specialized languages. Our approach also provides the mod-

eller with a modelling framework through the required model levels. This could also add structure and thereby perhaps even reduce the modelling effort. Our method comes with some predefined pragmatics which are of general use. But, our method is open for adding more pragmatics if need should be. Moreover, pragmatics can be used for adding more technical information which could be derived automatically. This way, it is possible to gradually extend the degree of automation of our method without changing the method itself.

Another objective of our method is the generation of code for different target languages and platforms. To this end, ATTs and template bindings were introduced; by replacing the templates and template binding, code for a different platform can be generated. In a way, a set of templates along with a template binding can be considered as a characterization of a target platform. And the code generator can be customized for different platforms by modifying templates. The concepts of principals and services in our approach, lend themselves nicely to the object oriented paradigm where principals can be realized as classes, and services can be realized as methods. The control-flow block structure fits well with imperative paradigm with loops and conditional statements. Therefore, it seems likely that it would be simple to create templates for languages and platforms with roots in these paradigms such as Java, Python and C. For functional languages and platforms, which do not have control flow structure such as loops and conditionals, this could be a little more difficult. A last objective of our method is the readability of the generated code. This might be a bit subjective, although some metrics exists [6]. With control blocks, ATTs, and templates reflecting these constructs in the target language, we try to emulate code written by human programmers. A detailed evaluation, however, is future work.

We have shown that our method works for a simple example and for one target platform. An evaluation for larger examples and other target platforms is future work. Likewise, we still need to show that the same CPN models can be used for verification as well as code generation. Though verification is not the main focus, future work will, at least, demonstrate that verification from the model is possible in principle. A first step towards verification was taken in [12].

References

1. M. Alanen, J. Lilius, I. Porres, and D. Truscan. On modeling techniques for supporting model driven development of protocol processing applications. In *Model-Driven Software Development*, pages 305–328. Springer, 2005.
2. C. Baier and J-P Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. J. Billington, G.E. Gallasch, and B. Han. A coloured Petri net approach to protocol verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, 2004.
4. Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.
5. S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3 – 23, 1987.

6. R.P.L. Buse and W.R. Weimer. A metric for software readability. In *Proc. of ISSTA'08*, pages 121–130, NY, USA, 2008. ACM.
7. C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification. In *Proc. of ICATPN'10*, volume 6128 of *LNCS*, pages 145–164. Springer, 2010.
8. M. Hannikainen, J. Knuutila, T. Hamalainen, and J. Saarinen. Using SDL for implementing a wireless medium access control protocol. In *proc International Symposium on Multimedia Software Engineering*, pages 229 –236, 2000.
9. K. Jensen. Coloured Petri nets and invariant methods. *Theoretical Computer Science*, 14:317–336, 1981.
10. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
11. L. M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G. E. Gallasch. Model-based development of a course of action scheduling tool. *International Journal on Software Tools for Technology Transfer*, 10:5–14, 2008.
12. L. M. Kristensen and K. I.F. Simonsen. Towards a CPN-based modelling approach for reconciling verification and implementation of protocol models. In *Proc. of MOMPES'12*, LNCS. Springer, 2012.
13. L. M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
14. L.M. Kristensen and K. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Proc. of Advanced Course on Petri Nets*, LNCS. Springer, 2012.
15. G. Leduc and F. Germeau. Verification of security protocols using LOTOS-method and application. *Computer Communications*, 23(12):1089 – 1103, 2000.
16. K. H. Mortensen. Automatic code generation method based on coloured Petri net models applied on an access control system. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 367–386, 2000.
17. J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. In *In Proc. TOOLS '00*, pages 82–93.
18. PetriCode. *Project Web Site*. http://kentis.github.io/nppn-cli/.
19. S. Philippi. Automatic code generation from high-level Petri-nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444 – 1455, 2006.
20. D. Pozza, R. Sisto, and L. Durante. Spi2Java: automatic cryptographic protocol Java code generation from spi calculus. In *Proc. of Advanced Information Networking and Applications*, volume 1, pages 400 – 405, 2004.
21. K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Code generation for protocols from CPN models annotated with pragmatics – extended abstract. In 24^{th} *Nordic Workshop on Programming Theory*, November 2012.
22. K. I.F Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocol Software from CPN models Annotated with Pragmatics. Technical Report IMM-Technical Reports-2013-01, Technical University of Denmark, DTU Informatics, January 2013. Available via `http://bit.ly/WwH2hf`.
23. I. Suzuki. Formal analysis of the alternating bit protocol by temporal Petri nets. *Software Engineering, IEEE Transactions on*, 16(11):1273 –1281, nov 1990.
24. K. Wagstaff, K. Peters, and L. Scharenbroich. From protocol specification to statechart to implementation. Technical Report CL08-4014, Jet Propulsion Laboratory, 2008.

Chapter 10

Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification

Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification

Kent Inge Fagerland Simonsen [*1,2], Lars Michael Kristensen [†1], and Ekkart Kindler [‡2]

[1]Department of Computing, Mathematics, and Physics, Bergen University College, Norway
[2]DTU Informatics, Technical University of Denmark, Denmark

August, 2014

DTU Compute Technical Report-2014-16

[*]Email: kifs@hib.no, kisi@imm.dtu.dk
[†]Email: lmkr@hib.no
[‡]Email: eki@imm.dtu.dk

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Building 324, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

DTU Compute Technical Reports: ISSN 1601-2321

Abstract

This paper presents the formal definition of Pragmatics Annotated Coloured Petri Nets (PA-CPNs). PA-CPNs represent a class of Coloured Petri Nets (CPNs) that are designed to support automated code generation of protocol software. PA-CPNs restrict the structure of CPN models and allow Petri net elements to be annotated with so-called pragmatics, which are exploited for code generation. The approach and tool for generating code is called PetriCode and has been discussed and evaluated in earlier work already. The contribution of this paper is to give a formal definition for PA-CPNs; in addition, we show how the structural restrictions of PA-CPNs can be exploited for making the verification of the modelled protocols more efficient. This is done by automatically deriving progress measures for the sweep-line method, and by introducing so-called service testers, that can be used to control the part of the state space that is to be explored for verification purposes.

1 Introduction

Although Coloured Petri Nets (CPNs) [2] have been widely used for modelling and verifying network protocols, rather limited research has been conducted into approaches that allow us to automatically generate the implementation of the protocols from the CPN models. And (to the best of our knowledge) there do not exist approaches that at the same time can be used for verification and code generation of network protocol software based on CPN models. In earlier work [5], we have presented an approach and a tool called PetriCode, which allowed us to automatically generate the protocol software from a restricted class of CPNs. One of the objectives of PetriCode was to be able to generate code for different platforms. Another main objective was that the used CPN models could still be applied for verifying the correctness of the network protocols.

The PetriCode approach uses a class of CPNs with a slightly restricted structure. On the one hand, these restrictions help making explicit the structure of the protocol, its principals, channels, and services. On the other hand, these restrictions make it possible to automatically generate code, the protocol software, from the CPNs modelling the protocol. One feature of this class of CPNs are so-called *pragmatics*, which are annotations to certain elements of the CPNs, which indicate the purpose of the respective modelling element and are exploited by the code generator. This way, models from which code can be generated are not cluttered with all kinds of technical information so that the same CPN models can be used for verification and code generation.

The PetriCode approach and tool have been presented, discussed and evaluated in earlier work already [5, 6]. In this paper, we formally define this restricted class of CPNs, which we call *Pragmatic Annotated CPNs* (*PA-CPNs*). In addition, we show that PA-CPNs are still amenable to verification, and that the structural restrictions on that class can actually make the verification more efficient: First, the structure of *PA-CPNs* allows us to automatically add so-called *service testers* to the model of the protocol, which reduce the state space of the model and, therefore, reduce the computation effort needed for verification. Second, the structural restrictions of *PA-CPNs* induce a natural progress measure that can be exploited by a verification technique that is called *sweep-line method* [1, 3], which again makes verification more efficient by reducing the

number of states that need to be stored at the same time in the verification tool. The formal definitions of PA-CPNs are illustrated by a running example, which is a simple framing protocol. By using this example, we also illustrate how the structure of PA-CPNs can exploited for verifying the protocol in a more efficient way.

For the rest of this paper, we assume that the reader is familiar with the basic concepts of Petri nets and high-level Petri nets in general. In Sect. 2, we introduce CPNs by an example and rephrase the standard definitions of CPNs [2]. The example used to explain CPNs in Sect. 2 is already a PA-CPN, but the specific structure mandated by PA-CPNs will first be discussed and formalized in Sect. 3 and Sect. 4: Section 3 covers the definitions concerning the specific pragmatics and restrictions of the different types of PA-CPN modules, and Sect. 4 formalizes one specific aspect, which makes sure that services can be represented by typical constructs for control flow. In Sect. 5, we discuss and formalize the extension of PA-CPNs with so-called service testers, which can be used for more efficiently verifying the model of the protocols. The actual verification by using the sweep-line method is discussed in Sect. 6. At last, in Sect. 7, we sum up the general findings and briefly discuss related work.

2 Protocol Example and Coloured Petri Nets

The definition of PA-CPNs relies on the definition of hierarchical CPNs given in [2]. Below we introduce the basic definitions and notations for hierarchical CPNs and the protocol CPN model that we will use as a running example throughout this paper. We present only the syntactical definition of hierarchical CPNs as PA-CPNs have the same semantics as ordinary hierarchical CPNs for simulation and verification purposes.

2.1 Protocol Example

The CPN model to be used as a running example models a protocol consisting of a *sender* and a *receiver* operating over an unreliable channel which may both re-order and loose messages. The sender sends messages tagged with sequence numbers to the receiver and waits for an acknowledgement for each message to be returned from the receiver before sending the next message. Hence, the protocol operates according to the stop-and-wait principle.

The CPN model of the protocol consists of eight hierarchically organised *modules*. Below we present selected modules of the CPN model used to illustrate the definition and verification techniques in this paper[1]. Figure 1 shows the top-level module consisting of three *substitution transitions* (drawn as double-bordered rectangles) and representing the Sender, the Receiver, and the Channel connecting them. The two *places* SenderChannel and ReceiverChannel model buffering communication endpoints connecting the sender and the receiver to the communication channel. The definition of the *colour set* (type) `Endpoint` determining the kind of tokens that can reside on these two places is provided in Fig. 2. Each of the three substitution transitions has an associated *submodule* indicated by the rectangular tag positioned next to the substitution transition.

[1]The complete CPN model is available via `www.petricode.org/examples/SWProtocol+driver.cpn`

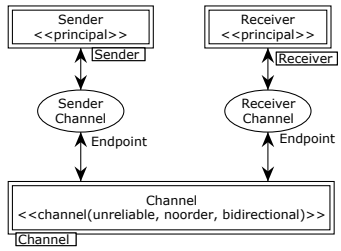


Figure 1: The top-level (prime) CPN module of the protocol model

```
colset Packet  = union DATA : Data + ACK : Ack;
colset EndpointId = INT;

colset ChannelPacket =
 record src : EndpointId * dest : EndpointId
        * packet : Packet;
colset ChannelPackets = list ChannelPacket;

colset Endpoint = record name : EndpointId *
inb : ChannelPackets *
outb : ChannelPackets;
```

Figure 2: Colour set (type) declarations used in Fig. 1

The annotations written in $\langle\langle\ \rangle\rangle$ are the *pragmatics* annotations that we formally introduce in the next section when defining PA-CPNs; they can be ignored for now.

Figure 3 shows the Sender module, which is the submodule associated with the Sender substitution transition in Fig 1 and defines the protocol for the Sender principal. The module has two substitution transitions modelling the main operations of the sender which is the sending of messages (substitution transition send) and the reception of acknowledgements (substitution transition receiveAck). The places ready, runAck, and nextSend are used to model the internal state of the sender. The place ready has an *initial marking* consisting of a token with the colour `()` (unit) which is the single value contained in the predefined colour set `UNIT`. This indicates that initially the sender is ready to perform a send operation. For a place with colour set `UNIT`, we omit (by convention) the specification of the colour set in the graphical representation. The place runAck, which has a boolean colour set, initially contains a token with the value `false` indicating that the sender is not initially in a state where it can receive acknowledgements. The place nextSend is used to keep track of the sequence of the message that the sender is currently sending. The place SenderChannel is a *port place* (indicated by the double border) and is used by the module to exchange tokens with its upper level module, which was shown

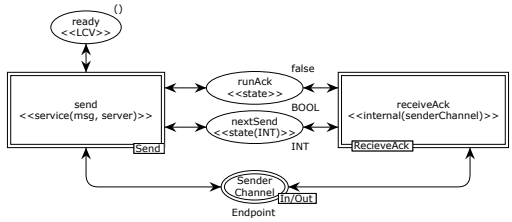


Figure 3: The sender CPN module

in Fig. 1. In this case, SenderChannel is an *input-output port place* as specified by the In/Out tag positioned next to the place. The place is associated with the SenderChannel *socket place* in Fig. 1 which means that any tokens removed (added) from (to) this place in the Sender module will also be reflected in the Protocol module.

Figure 4 shows the Send module which is the submodule of the send substitution transition in Fig. 3. This submodule models the sending of a list of messages from the sender to the receiver. The port places ready, SenderChannel, nextSend, and runAck are associated with the accordingly named socket places in the module shown in Fig. 3. The list of messages to be sent is provided via the place message (top) annotated with the `driver` pragmatic. This place is a *fusion place* as indicated by the rectangular tag positioned next to the place. The name inside the tag specifies the *fusion set* that the place belongs to. A fusion set is a set of places with the property that when tokens are removed (added) to one place in the set, then the token will be removed (added) to all members. Conceptually, all the places of a fusion set are merged into a single compound place. The place end (at the bottom) annotated with a `driver` pragmatic is also a member of a fusion set. These fusion sets are used to connect PA-CPNs to test driver modules to be introduced later; these places are, formally, not part of the service level module or the complete protocol. The places annotated by the `driver` pragmatic are used by the test driver module to control the order and the parameters of the invocation of the services of the protocol during the verification of the protocol (see Sect. 5 and Sect. 6). The code generator ignores these places since, in the actual protocol software, the services of the protocol are invoked externally; the order of invocation of the services and the parameters are determined by the protocol's environment.

The sending of a list of messages starts with the occurrence of the transitions send, which places the list of messages to be sent on place message, puts a token on the place nextSend corresponding to the first sequence number, and a token on place runAck to indicate that acknowledgements can now be received. The place limit is used to put an upper bound on the number of attempts to retransmit a message when the transmission fails. After an occurrence of transition send, transition sendMsg may occur sending a message by putting it in the output buffer modelled by the place SenderChannel. The *guard* used on the transition sendMsg (by convention written in square brackets next to the transition) ensures that the data being sent matches the sequence number of the message currently being sent. If the retransmission limit is reached, the sender will stop as modelled by the transition return putting a token on place end. If

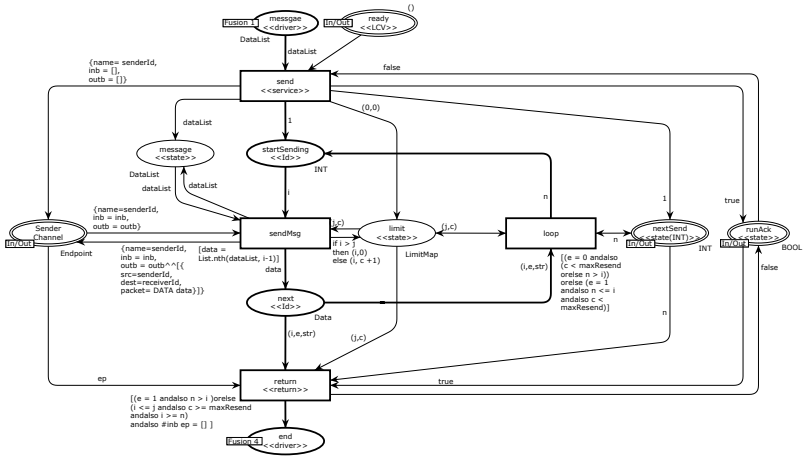


Figure 4: The Send module

```
colset LimitMap = product INT * INT;
colset Data = product  INT * INT * STRING;
colset DataList = list Data;

val maxResend = 2;
var i, j, k, l, c, e, n :INT;
var data : Data;
var dataList : DataList;
var str : STRING;
```

Figure 5: Colour set (type) declarations used in Fig. 4

the retransmission limit is not reached for the current message, the transition loop will put a token back on startSending such that the next message can be sent. The colour set definitions and *variables* used in the inscriptions of Fig. 4 are provided in Fig. 5.

Above, we have presented the example CPN model that will be used as a running example throughout this paper, and we have informally introduced the hierarchical constructs of CPNs in the form of modules, substitution transitions, port and socket places, and fusion places.

2.2 Formal Definitions of Hierarchical CPNs

In this subsection, we formally define hierarchical CPNs as the later formal definition of PA-CPNs will be based on the formal definition of hierarchical CPNs. Definition 2.1 provides the formal definition of CPN modules. In the definition, we use $Type[v]$ to denote the type of a variable v, and we use $EXPR_V$

to denote the set of expressions with free variables contained in a set of variables V. For an expression e containing a set of free variables V, we denote by $e\langle b\rangle$ the result of evaluating e in a binding b that assigns a value to each variable in V. We use $\mathit{Type}[e]$ for an expression e (an arc expression, a guard, or an initial marking) to denote the the type of e. For a non-empty set S, we use S_{MS} to denote the type corresponding to the set of all multi-sets over S.

Definition 2.1. A **Coloured Petri Net Module** (Def. 6.1 in [2]) is a tuple $CPN_M = (CPN, T_{\mathsf{sub}}, P_{\mathsf{port}}, PT)$, such that:

1. $CPN = (P, T, A, \Sigma, V, C, G, E, I)$ is a Coloured Petri Net (Def. Y in [2]) where:
   (a) P is a finite set of **places** and T is a finite set of **transitions** T such that $P \cap T = \emptyset$.
   (b) $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed **arcs**.
   (c) Σ is a finite set of non-empty **colour sets** and V is a finite set of **typed variables** such that $\mathit{Type}[v] \in \Sigma$ for all variables $v \in V$.
   (d) $C : P \to \Sigma$ is a **colour set function** that assigns a colour set to each place.
   (e) $E : A \to \mathit{EXPR}_V$ is an **arc expression function** that assigns an arc expression to each arc a such that $\mathit{Type}[E(a)] = C(p)_{MS}$, where p is the place connected to the arc a.
   (f) $G : T \to \mathit{EXPR}_V$ is a **guard function** that assigns a guard to each transition t such that $\mathit{Type}[G(t)] = \mathit{Bool}$.
   (g) $I : P \to \mathit{EXPR}_\emptyset$ is an **initialisation function** that assigns an initialisation expression to each place p such that $\mathit{Type}[I(p)] = C(p)_{MS}$.
2. $T_{\mathsf{sub}} \subseteq T$ is a set of **substitution transitions**.
3. $P_{\mathsf{port}} \subseteq P$ is a set of **port places**.
4. $PT : P_{\mathsf{port}} \to \{\mathsf{IN}, \mathsf{OUT}, \mathsf{I/O}\}$ is a **port type function** that assigns a port type to each port place.

□

Socket places are not defined explicitly as part of a module because they are implicitly given via the arcs connected to the substitution transition. For a substitution transition t, we denote by $ST(t)$ a mapping that maps each socket place p into its type, i.e., $ST(t)(p) = \mathsf{IN}$ if p is an input socket, $ST(t)(p) = \mathsf{OUT}$ if p is an output socket, and $ST(t)(p) = \mathsf{I/O}$ if p is an input/output socket.

The definition of a hierarchical CPN is provided below. A hierarchical CPN consists of a set of disjoint CPN modules, a submodule function assigning a (sub)module to each substitution transition, and a port-socket relation that associates port places in a submodule to the socket places of its upper layer module. The set of socket places for a substitution transition t are the place connected to the substitution transition and is denoted by $P_{\mathsf{sock}}(t)$. The definition requires that the module hierarchy (to be defined in Def. 2.3) is acyclic in order to ensure that there are only a finite number of instances of each module. Furthermore, port and socket places can only be associated with each other, if they have the same colour set and the same initial marking.

Definition 2.2. A **hierarchical Coloured Petri Net** (Def. 6.2 in [2]) is a four-tuple $CPN_H = (S, SM, PS, FS)$ where:

1. S is a finite set of **modules**. Each module is a **Coloured Petri Net Module** $s = ((P^s, T^s, A^s, \Sigma^s, V^s, C^s, G^s, E^s, I^s), T^s_{\mathsf{sub}}, P^s_{\mathsf{port}}, PT^s)$. It is required that $(P^{s_1} \cup T^{s_1}) \cap (P^{s_2} \cup T^{s_2}) = \emptyset$ for all $s_1, s_2 \in S$ with $s_1 \neq s_2$.
2. $SM : T_{\mathsf{sub}} \to S$ is a **submodule function** that assigns a **submodule** to each substitution transition. It is required that the module hierarchy (see Definition 2.3) is acyclic.
3. PS is a **port–socket relation function** that assigns a **port–socket relation** $PS(t) \subseteq P_{\mathsf{sock}}(t) \times P^{SM(t)}_{\mathsf{port}}$ to each substitution transition t. It is required that $ST(t)(p) = PT(p')$, $C(p) = C(p')$, and $I(p)\langle\rangle = I(p')\langle\rangle$ for all $(p, p') \in PS(t)$ and all $t \in T_{\mathsf{sub}}$.
4. $FS \subseteq 2^P$ is a set of non-empty and disjoint **fusion sets** such that $C(p) = C(p')$ and $I(p)\langle\rangle = I(p')\langle\rangle$ for all $p, p' \in fs$ and all $fs \in FS$.

□

The module hierarchy of a hierarchical CPN model is a directed graph with a node for each module and an arc leading from one module to another module if the latter module is a submodule of one of the substitution transitions of the former module. In the definition, T_{sub} denotes the union of all substitution transitions of the hierarchical CPN, and T^s_{sub} denotes all substitution transitions in a module s.

Definition 2.3. The **module hierarchy** for a hierarchical Coloured Petri Net $CPN_H = (S, SM, PS, FS)$ is a directed graph $MH = (N_{MH}, A_{MH})$, where

1. $N_{MH} = S$ is the set of **nodes**.
2. $A_{MH} = \{(s_1, t, s_2) \in N_{MH} \times T_{\mathsf{sub}} \times N_{MH} \mid t \in T^{s_1}_{\mathsf{sub}} \wedge s_2 = SM(t)\}$ is the set of **arcs**.

The roots of MH are called **prime modules**, and the set of all prime modules is denoted S_{PM}.

□

3 Pragmatic Annotated CPNs

PA-CPNs mandates a particular structure of the CPN models and enables the CPN elements to be annotated with *pragmatics* used to direct the automated code generation. In a PA-CPN, the modules of the CPN model are required to be organised into three levels referred to as the *protocol system level*, the *principal level*, and the *service level*. In a PA-CPN, it is required that there exists exactly one prime module. This prime module represents the protocol system level. The Protocol module shown in Fig. 1 comprises the protocol system level of the PA-CPN model of the framing protocol; it specifies the protocol principals

in the system and the channels connecting them. The substitution transitions representing principals are specified using the `principal` pragmatic, and the substitution transitions representing channels are specified using the `channel` pragmatic. In the CPN model, pragmatics are shown by annotations enclosed in guillemets. On the principal level, there is one module for each principal of the protocol as defined on the protocol system level. The framing protocol has two modules at the principal level corresponding to the sender and the receiver. Figure 3 shows the principal level module for the sender. A principal level module is required to model the services that the principal is providing, and the internal states and life-cycle of the principal. For the sender, there are two services, which are indicated by the `service` pragmatics: send and receiveAck. Substitution transition representing services that can be externally invoked are specified using the `service` pragmatic, whereas services that are to be invoked only internally are specified using the `internal` pragmatic. The service level modules model the behaviour of the individual services. The module in Fig. 4 is an example of a module at the service level modelling the send service provided by the sender principal.

We formally define PA-CPNs as a tuple consisting of a hierarchical CPN, a protocol system module (PSM), a set of principal level modules (PLMs), a set of service level modules (SLMs), a set of channel modules (CHMs), and a structural pragmatics mapping (SP) that maps substitution transitions into structural pragmatics and capturing the annotation of the substitution transitions. It should be noted that since channel modules do not play a role in the code generation but are only a CPN model artifact used to connect the principals for simulation purposes, we do not impose any specific requirements to the internal structure of channel level modules.

Definition 3.1. A **Pragmatics Annotated Coloured Petri Net** (PA-CPN) is a tuple $CPN_{PA} = (CPN_H, PSM, PLM, SLM, CHM, SP)$, where:

1. $CPN_H = (S, SM, PS, FS)$ is a hierarchical CPN.
2. $PSM \in S$ is a **protocol system module** (see Def. 3.2) and the only prime module of CPN_H.
3. $PLM \subseteq S$ is a set of **principal level modules** (see Def. 3.3).
4. $SLM \subseteq S$ is a set of **service level modules** (see Def. 3.4).
5. $CHM \subseteq S$ is a set of **channel modules**.
6. PSM, PLM, SLM, CHM constitute a partition of S.
7. $SP : T_{sub} \to \{\texttt{principal}, \texttt{service}, \texttt{internal}, \texttt{channel}\}$ is a **structural pragmatics mapping** such that:
   (a) Substitution transitions annotated with a `principal` pragmatic have an associated principal level module:
   $\forall t \in T_{sub} : SP(t) = \texttt{principal} \Rightarrow SM(t) \in PLM$
   (b) Substitution transitions annotated with a `service` or `internal` pragmatic are associated with a service level module:
   $\forall t \in T_{sub} : SP(t) = \texttt{service} \wedge SP(t) = \texttt{internal} \Rightarrow SM(t) \in SLM$

(c) Substitution transitions annotated with a `channel` pragmatic are associated with a channel module:
$\forall t \in T_{sub} : SP(t) = \texttt{channel} \Rightarrow SM(t) \in CHM$

□

The protocol system module (PSM) models the principals of the protocol and the channels that connects them. The PSM module is defined as a tuple consisting of a CPN module and a pragmatic mapping PM that associates a pragmatic to each substitution transition. The requirement to the module is that all substitution transitions must be substitution transitions and annotated with either a `principal` or a `channel` pragmatic. Furthermore, two substitution transitions representing principals cannot be connected only by a place, there must be a substitution transition representing a channel in between. This reflects the fact that it is possible for principals to communicate via channels only[2].

Definition 3.2. A **Protocol System Module** of a PA-CPN with a structural pragmatics mapping SP_{PA} is a tuple $CPN_{PSM} = (CPN^{PSM}, PM)$, where:

1. $CPN^{PSM} =$
$((P^{PSM}, T^{PLM}, A^{PSM}, \Sigma^{PSM}, V^{PSM}, C^{PSM}, G^{PSM}, E^{PSM}, I^{PSM}),$
$T^{PSM}_{sub}, P^{PSM}_{port}, PT^{PSM})$ is a CPN module (see Def. 2.1).
2. All transitions are substitution transitions $T^{PSM} = T^{PSM}_{sub}$.
3. $PM : T^{PSM}_{sub} \rightarrow \{\texttt{principal}, \texttt{channel}\}$ is a **pragmatics mapping** satisfying:
   (a) All substitution transitions are annotated with either `principal` or `channel` pragmatic: $\forall t \in T^{PSM}_{sub} : PM(t) \in \{\texttt{principal}, \texttt{channel}\}$.
   (b) The pragmatics mapping must coincide with the structural pragmatic mapping of PA-CPN: $\forall t \in T^{PSM}_{sub} : PM(t) = SP(t)$.
   (c) All places are connected to at most one substitution transition annotated with principal and at most one annotated with channel:
   $\forall p \in P^{PSM} : \forall t_1, t_2 \in X(p) : PM(t_1) = PM(t_2) \Rightarrow t_1 = t_2$.

□

A principal level module specifies the services provided by a principal and is defined as a tuple consisting of a CPN module and a principal level pragmatic mapping PLP. Each service is represented by a substitution transition which can be annotated with either a service or internal pragmatic depending on whether the service is visible externally or not. The non-port places of a principal level model can be annotated with either a `state` or an `LCV` pragmatic. Places annotated with a `state` pragmatic represent internal states of the principal, whereas places annotated with an `LCV` pragmatic represent the life-cycle of the principal by putting restrictions on the order in which services can be invoked. As an example, the place ready in Fig. 3 ensures that only one message at a time is sent using the send service.

[2] In the definition, we use $X(p)$ to denote the set of transitions connected to a place p.

Definition 3.3. A **Principal Level Module** of a PA-CPN with a structural pragmatics mapping SP_{PA} is a tuple $CPN_{PLM} = (CPN_{PLM}, T^{PLM}_{sub}, P^{PLM}_{port}, PT^{PLM}, PLP)$ where:

1. $CPN_{PLM} = (P^{PLM}, T^{PLM}, A^{PLM}, \Sigma^{PLM}, V^{PLM}, C^{PLM}, G^{PLM}, E^{PLM}, I^{PLM})$ is a CPN module (see Def. 2.1).
2. All transitions are substitution transitions: $T^{PLM} = T^{PLM}_{sub}$
3. $PLP : T^{PSM}_{sub} \cup P^{PLM} \setminus P^{PLM}_{port} \rightarrow \{\texttt{service}, \texttt{internal}, \texttt{state}, \texttt{LCV}\}$ is a **principal level pragmatics mapping** satisfying:
   (a) All non-port places are annotated with either a `state` or a `LCV` pragmatic: $\forall p \in P^{PLM} \setminus P^{PLM}_{port} \Rightarrow PLP(p) \in \{\texttt{state}, \texttt{LCV}\}$
   (b) All substitution transitions are annotated with a `service` or `internal` pragmatic: $\forall t \in T^{PSM}_{sub} : PLP(t) \in \{\texttt{service}, \texttt{internal}\}$.

□

It should be noted that we do not associate any pragmatics with the port place of the module as it follows from the definition of the protocol system module that a port place in a principal level module can only be associated with a socket place connected to a channel substitution transition.

The service level modules specify the detailed behaviour of the individual services and constitute the lowest level modules in a PA-CPN model. In particular, there are no substitution transitions in modules at this level. The Send module in Fig. 4 is an example of a module at the service level. It models the behaviour of the send service in a control-flow oriented manner. The control-flow path, which defines the control flow of the service, is made explicit via the use of the `Id` pragmatics. The entry point of the service is indicated by annotating a single transition with a `service` pragmatic, and the exit (termination) point of the service is indicated by annotating a single transition with a `return` pragmatic. In addition, a non-port place can be annotated with a state pragmatic to indicate that this place models a local state of the service. The `driver` is used by service tester modules to facilitate verification by reducing the state space of the protocol model. The places annotated with an `Id` pragmatic determine a subset of the module, which we call the *underlying control-flow net*; and it is required that this net is block decomposable (which will be defined later in Sect. 4) in order to support a natural translation into programming language control flow structures. In Fig. 4, the underlying control flow net is highlighted via the places, transitions, and arcs having a thick border. A service level module is defined as consisting of a CPN module without substitution transitions and with a service level pragmatic mapping that associates pragmatics to the model elements as described above.

In the definition below we use the notation $\exists! x \in X : p(x)$ to denote that there exists exactly one element x in a set X satisfying a predicate p – i. e. the element x is uniquely characterized by property $p(x)$.

Definition 3.4. A **Service Level Module** is a tuple $CPN_{SLM} = (CPN_{SLM}, T^{SLM}_{sub}, P^{SLM}_{port}, PT^{SLM}, SLP, SLT)$ where:

1. $CPN_{SLM} = (P^{SLM}, T^{SLM}, A^{SLM}, \Sigma^{SLM}, V^{SLM}, C^{SLM}, G^{SLM}, E^{SLM}, I^{SLM})$ is a CPN module (see Def. 2.1)
2. There are no substitution transitions: $T^{SLM}_{sub} = \emptyset$.
3. $SLP : T^{PLM} \cup P^{PLM} \setminus P^{PLM}_{port} \rightarrow \{\texttt{Id}, \texttt{state}, \texttt{service}, \texttt{return}, \texttt{driver}\}$ is a **service level pragmatic mapping** satisfying:
   (a) Each place is either annotated with `Id`, `state`, `driver` or is a port place : $\forall p \in P^{SLM} \setminus P^{SLM}_{port} : SLP(p) = \texttt{Id} \vee SLP(p) = \texttt{state} \vee SLP(p) = \texttt{driver}$
   (b) There exits exactly one transition annotated with `service`: $\exists! t \in T^{SLM} : SLP(t) = \texttt{service}$
   (c) There exits exactly one transition annotated with `return`: $\exists! t \in T^{SLM} : SLP(t) = \texttt{return}$
4. For all $t \in T^{SLM}$ and $p \in P^{SLM}$ we have:
   (a) Transitions consume one token from input places annotated with an `Id` pragmatic: $(p,t) \in A^{SLM} \wedge SLP(p) = \texttt{Id} \Rightarrow |E(p,t)\langle b \rangle| = 1$ for all bindings b of t.
   (b) Transitions produce one token on output places annotated with an `Id` pragmatic: $(t,p) \in A^{SLM} \wedge SLP(p) = \texttt{Id} \Rightarrow |E(t,p)\langle b \rangle| = 1$ for all bindings b of t.
   (c) Only transitions annotated with a `service` pragmatic can have input places annotated with a `driver` pragmatic: $(p,t) \in A^{SLM} \wedge SLP(p) = \texttt{driver} \Rightarrow SLP(t) = \texttt{service}$
   (d) Only transitions annotated with a `return` pragmatic can have output places annotated with a `driver` pragmatic: $(t,p) \in A^{SLM} \wedge SLP(p) = \texttt{driver} \Rightarrow SLP(t) = \texttt{return}$
5. The underlying control flow block of CPN_{SLM} (Def. 4.2) is tree decomposable (Def. 4.4).

□

4 Block Decomposition of Control Flow Nets

In this section, we define formally when the control flow of a service level module is decomposable into blocks. Figure 6 shows the underlying control flow net of the service level module for the send operation of the sender from Fig. 4, which is a loop construct, basically.

In order to formally define the block decomposition, we need to define blocks first: these are Petri nets with a fixed entry and exit place. Then, we define the underlying control flow net of a service module. At last, we define when a block is decomposable into blocks, which represent the control flow constructs.

Figure 7 shows a graphical representation of a block in general, where the start and end place of the block are graphically represented by arcs from resp. to the border of the block.

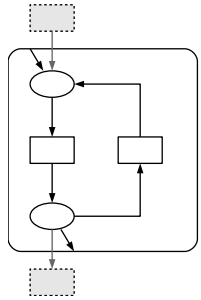


Figure 6: Decomposition of the control flow net of module SenderSend

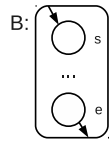


Figure 7: Graphical representation of a block.

Definition 4.1 (Block, atomic non-returning block)**.**
Let $N = (P, T, A)$ be a Petri net and $s, e \in P$. Then $B = (P, T, A, s, e)$ is called a *block* with *entry* s and *exit* e.

The block is *atomic*, if $P = \{s, e\}$, $s \neq e$, $|T| = 1$ and for $t \in T$, we have ${}^\bullet t = \{s\}$ and $t^\bullet = \{e\}$.

The block has a *safe entry*, if $s \neq e$ and ${}^\bullet s = \emptyset$ (i. e. the block will not return a token to the start place itself). The block has a *safe exit*, if $s \neq e$ and $e^\bullet = \emptyset$ (i. e. the block does not use a token from the end place itself).

An atomic block consists of a single transition, as shown in Fig. 10 later. For visualizing blocks with safe entry and safe exit, we introduce an additional graphical notation, which is shown in Fig. 8. The crossed out arc from within the block to the start place indicates that the block itself does not return a token to the entry place (safe entry); the crossed out arc from the end place to the interior of the block indicates that the block itself does not remove a token from its exit place (safe exit).

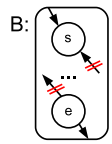


Figure 8: Graphical representation of safe entry and safe exit.

For easing the following definitions, we introduce an additional notation: For a block B_i, we refer to its constituents by $B_i = (P_i, T_i, A_u, s_i, e_i)$ without explicitly naming them every time.

The block that is underlying a service level model is basically obtained by all the places that are annotated with the `Id` pragmatics and the transitions in their pre- and postsets. The unique transition with the `service` pragmatics defines the entry place, and the unique transition with the `return` defines the exit place of this block; note that for technical reasons, these two transitions are not part of the block. Therefore, these transitions are shown by dashed lines in Fig. 6

Definition 4.2. [Underlying control flow net of SLM] Let CPN_{SLM} be a service level module as defined in Def. 3.4. Let $P = \{p \in P^{SLM} \setminus P^{SLM}_{port} | SLP(p) = Id\}$, let $T = T^{SLM} \cap^{\bullet} P \cap P^{\bullet}$, and let $A = A^{SLM} \cap ((T \times P) \cup (P \times T))\}$; moreover, let $s \in P$ be the unique place such that there exists a transition $t \in T = T^{SLM}$ with $(t, s) \in A^{SLM}$ and $SLP(t) = \texttt{service}$, and let $e \in P$ be the unique place e such that there exists a transition $t \in T = T^{SLM}$ with $(e, t) \in A^{SLM}$ and $SLP(t) = \texttt{return}$.

Then, we call $N = (P, T, A, s, e)$ the **underlying control flow net** of CPN_{SLM}.

The control flow of the generated code will be obtained by decomposing the underlying control flow net of a service level module into sub-blocks, which represent the control flow constructs. Note that we define the decomposition in a very general way at first, which does not yet restrict the possible control constructs. The decomposition into blocks, just makes sure that all parts of the block are covered by sub-blocks and that they overlap on entry and exit places only. In a second step, the decomposition is restricted in such a way that the decomposition captures certain control flow constructs (Def. 4.4).

Definition 4.3 (Decomposition of a block)**.**
Let $B = (N, s, e)$ be a block with net $N = (P, T, F)$.

A set of blocks $B_1, \ldots, B_n$ is called a decomposition of block B, if the following conditions are met:

1. The sub-blocks contain only elements from B, i. e. for each $i \in \{1, \ldots, n\}$, we have $P_i \subseteq P$, $T_i \subseteq T$, and $F_i \subseteq F \cap ((P_i \times T_i) \cup (T_i \times P_i))$.
2. The sub-blocks contain all elements of B, i. e. $P = \bigcup_{i=1}^{n} P_i$, $T = \bigcup_{i=1}^{n} T_i$, and $F = \bigcup_{i=1}^{n} F_i$
3. The inner structure of all sub-blocks are disjoint, i. e. for each $i, j \in \{1, \ldots, n\}$ with $i \neq j$, we have $T_i \cap T_j = \emptyset$ and $P_i \cap P_j = \{s_i, e_i\} \cap \{s_j, e_j\}$.

Note that, in some cases, two consecutive blocks should be safe, which means that either the exit of the preceding block is safe, or the entry of the succeeding block is safe or both. We represent this graphically as shown in Fig. 9.

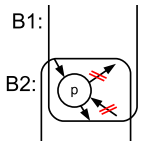


Figure 9: Safe join of two consecutive blocks

At last, we define when a decomposition of a block reflects some control flow construct. Note that this definition does not only define decomposability into control constructs; it also defines a tree structure which reflects the control structure of the block. The formal definition is illustrated in Fig. 10.

Definition 4.4 (Tree decompositions of a block)**.**

The *block trees* associated with a block are inductively defined:

- If B is an atomic block, then the tree with the single node $B : atomic$ is a *block tree* associated with B.
- If B is a block and B_1 and B_2 is a decomposition of B, and for some X, $B_1 : X$ is a block tree associated with B_1, and $B_2 : atomic$ is a block tree associated with B_2, and if B_1 has a safe entry and a safe exit and $s_1 = s$, $e_1 = e$, $s_2 = e$, and $e_2 = s$, then the tree with top node $B : loop$ and the sequence of sub-trees $B_1 : X$ and $B_2 : atomic$ is a *block tree* associated with B.
- If B is a block and for some n with $n \geq 2$ the set of blocks $B_1, \ldots, B_n$ is a decomposition of B, and have a safe entry and a safe exit, and $B_1 : X_1, \ldots, B_n : X_n$ for some $X_1, \ldots, X_n$ are block trees associated with $B_1, \ldots, B_n$, and if for every $i \in \{1, \ldots, n\}$ we have $s_i = s$ and $e_i = e$, then the tree with top node $B : choice$ with the sequence of sub-trees $B_i : X_i$ is a *block tree* associated with B.
- If B is a block and for some n with $n \geq 2$ the set of blocks $B_1, \ldots, B_n$ is a decomposition of B, and, for some $X_1, \ldots, X_n$, the trees $B_1 : X_1, \ldots, B_n : X_n$ are block trees associated with $B_1, \ldots, B_n$, and if there exist different places $p_0, \ldots, p_n \in P$ such that $s = p_0$, $e = p_n$, and for each $i \in \{0, \ldots, n-1\}$ we have $s_i = p_i$, $e_i = p_{i+1}$, and B_i has a safe exist or B_{i+1} has a safe entry, then the tree with top node $B : sequence$ and the sequence of sub-trees $B_i : X_i$ is a *block tree* associated with B.

Note that the tree decomposition of a block is not necessarily unique. For example a longer sequence of atomic blocks could be decomposed into different junks. The reason is that sequences can have arbitrary length according to our definition, which makes the definitions much more elegant and allows us to have long sequences in a single sequence construct. The tool actually resolves this ambiguity by making blocks with a sequence as large as possible.

Fig. 4 is an example of an SLM. Its underlying control flow net was shown in Fig. 6. This block is decomposed in a loop, which in turn consists of an atomic block. The service transition itself as well as the return transition are actually not part of the underlying control flow net.

5 Service Testers

The service level modules constitute the active part of a PA-CPN model, and the execution of the individual service provided by a principal starts at the transition annotated with a `service` pragmatic. The transitions annotated with a service pragmatic typically has a number of parameters which need to be

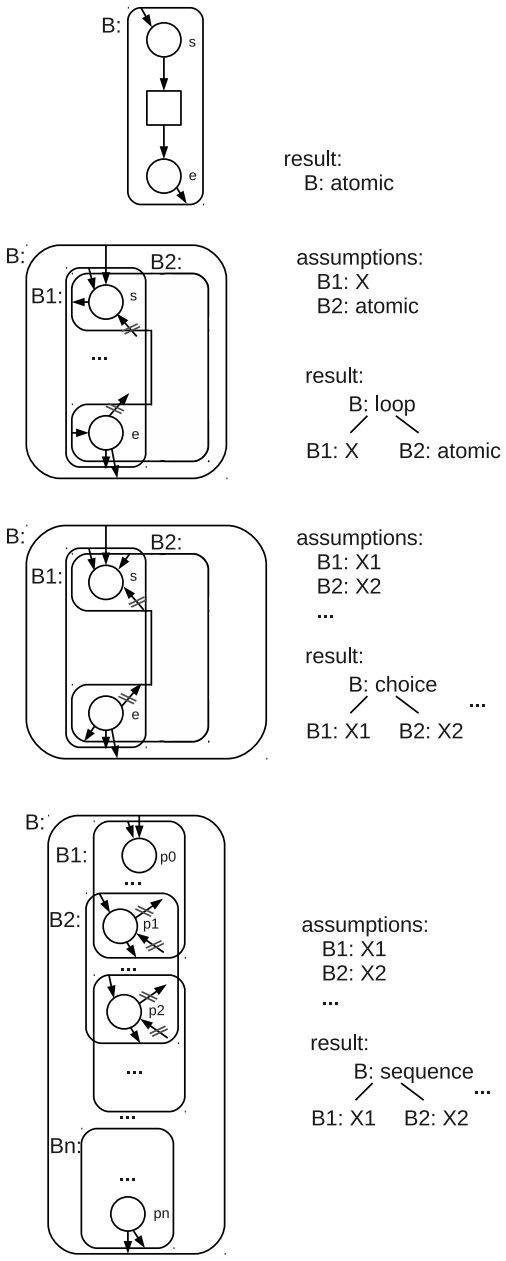


Figure 10: Inductive definition of block trees

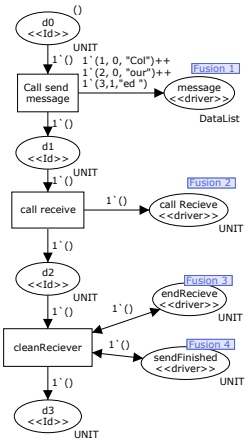


Figure 11: The Service tester

bound to values in order for the transition to occur. An example of this is the Send service transition in Fig. 4 which has dataList is a parameter. This means that there are often an infinite number of bindings for a service transition. To control the execution of a PA-CPN model when conducting validation by means of simulation and verification by means of state space exploration, we introduce the concept of *service tester modules* which can be used to guide the validation and verification process and represent a user of the service provided by the principal modules. An added advantage of service testers is that they can further contribute to reducing the state space during verification and progress measures can be automatically computed and used in conjunction with the sweep-line method for state-space exploration as will be explained in Sect. 6.

The service tester modules will be connected to the rest of the PA-CPN model through fusion places, and the service tester modules invokes the service provided by the principal by putting tokens on these places. Similarly, the service tester also receives any results from the invoked services via tokens on these fusion places. The fusion places used to connect service level modules and service tester modules are the only way fusion places are used on top of the concepts of PA-CPNs. In addition to the fusion places, a service tester module has an explicit control flow path similar to service level modules and `Id` pragmatics are used to make this explicit.

Figure 11 shows a server tester module for the PA-CPN model introduced in Sect. 3. The service tester drives the execution of a CPN model through fusion places. A service tester module is allowed to have only a single `Id` place that initially contains a token. In the case of Fig. 11, this is the place d0. The test driver first invokes the send service in Fig. 4 by putting a token in the fusion place message. Next, the service tester invokes the receive service in the receiver principal.

Below we formalise the control flow part of a service tester. In the definition, we use $I(p)\langle\rangle$ to denote the result of evaluating the initial marking expression $I(p)$ of a place p.

Definition 5.1. A **Service Tester Module** is a tuple $CPN_{STM} = (CPN_{STM}, T^{STM}_{sub}, P^{SLM}_{port}, PT^{SLM}, TPM)$ where:

1. $CPN_{SLM} = (P^{SLM}, T^{SLM}, A^{SLM}, \Sigma^{SLM}, V^{SLM}, C^{SLM}, G^{SLM}, E^{SLM}, I^{SLM})$ is a CPN module (see Def. 2.1)
2. There are no substitution transitions: $T^{STM}_{sub} = \emptyset$.
3. $TPM : P^{STM} \rightarrow \{\texttt{Id}, \texttt{driver}, \texttt{LCV}\}$ is a **service tester pragmatic mapping**.
4. $\exists! p \in I$ such that $|I^{STM}(p)\langle\rangle| = 1$.
5. For all $t \in T^{SLM}$ and $p \in P^{SLM}$ we have:
   (a) Transitions consume one token from input places annotated with an `Id` pragmatic: $(p,t) \in A^{SLM} \wedge TPM(p) = \texttt{Id} \Rightarrow |E(p,t)\langle b\rangle| = 1$ for all bindings b of t.
   (b) Transitions produce one token on output places annotated with an `Id` pragmatic: $(t,p) \in A^{SLM} \wedge TPM(p) = \texttt{Id} \Rightarrow |E(t,p)\langle b\rangle| = 1$ for all bindings b of t.
6. Transitions and places annotated with a `LCV` pragmatic must be connected with a double arc:
   $\forall p \in P^{STM}, t \in T^{SLM} : TPM(p) = \texttt{LCV} \Rightarrow ((t,p) \in A \Leftrightarrow (p,t) \in A)$
7. The underlying control flow block of CPN_{STM} (Def. 4.2) is tree decomposable (Def. 4.4)

□

As explained above, the idea is that a set of service tester modules can be connected to a PA-CPN by means of fusion places in order to control the execution of the services. Formally, we therefore define a PA-CPN equipped with service tester modules as a hierarchical CPN consisting of a set of modules that constitute a PA-CPN according to Def. 3.1 and a set of service tester modules which all constitute prime modules. Furthermore, we require that fusion places are connecting the service level modules and the service tester module so that they correspond to the invocation of services and collecting of a results from an executed service.

Definition 5.2. A **Pragmatics Annotated Coloured Petri Net with Service Testers** is tuple $CPN_{PAT} = (CPN_H, PSM, PLM, SLM, CHM, SP, STM)$, where:

1. $CPN_H = (S, SM, PS, FS)$ is a hierarchical CPN.
2. $CPN_{PAT} = (CPN_H, PSM, PLM, SLM, CHM, SP)$ is a PA-CPN
3. $STM \in S$ is a set of service tester modules all of which are prime modules.
4. The following conditions hold for all fusion sets $fs \in FS$:

(a) Places in a fusion set are either all annotated with a `driver` pragmatic or all annotated with a `LCV` pragmatic.

(b) A fusion set containing places with `driver` pragmatics can only contain places from a single service layer module and a single service tester module.

(c) A fusion set containing places with `LCV` pragmatics can only contain places related via a port-socket relationship or places belonging to service tester modules.

(d) If $p \in fs$ belongs to a service level module and has an output arc to a transition with a `service` pragmatic, then all places $p_t \in fs$ belonging to a service tester module STM have only input arcs from transitions in STM.

(e) If $p \in fs$ belongs to a service level module and has an input arc to a transition with a `return` pragmatic, then all places $p_t \in fs$ belonging to a service tester module STM can have output arcs to transitions in STM only.

□

6 Verification

State space exploration is the main verification method for CPNs and is based on the idea of explicitly enumerating the set of reachable states of the CPN model. Generally, this approach is limited by the available memory since the states need to be stored while the state space is generated. A large collection of techniques have been developed in order to alleviate this inherent complexity problem. In this section, we show how the sweep-line method [1, 3] can be used to alleviate the state explosion problem when conducting verification of PA-CPNs with service testers.

6.1 The Sweep-Line Method

The basic idea of the sweep-line method is to exploit a notion of *progress* exhibited by many systems. Exploiting progress makes it possible to explore all reachable states while storing only small subsets of the state space in memory at a time. This way, much larger state spaces can be investigated since never all states need to be stored at the same time. The additional structure imposed on CPNs by PA-CPNs and services testers means that PA-CPN models have several potential sources of progress that can be exploited by the sweep-line method. The control-flow in the service modules is one source of progress as there is a natural progression from the entry point of the service towards the exit point of the service. The life-cycle of a principal is another potential source of progress as there will often be an overall intended order in which the services provided by a principal is to be invoked, and this will be reflected in the life-cycle variables of the principal. Finally, the service testers are also a source of progress as a service tester will inherently progress from the start of the test towards the end of the test.

The subsets of states stored are determined via a *progress value* assigned to each state, and the method explores the states in a least-progress-first order. The sweep-line method explores states with a given progress value before progressing to the states with a higher progress value. When the method proceeds to consider states with a higher progress value, it deletes the states with a lower progress value from memory. The basic idea is to optimistically assume that the system does not *regress*, and hence states with a lower progress value will not be visited again and do not need to be kept in memory. If it turns out that the system regresses, then the method will mark states at the end of *regress edges* as *persistent* (i. e., store them permanently in memory) in order to ensure termination. In the presence of regression, the sweep-line method may visit some states multiple times. The fact that the sweep-line method deletes states means that verification of properties needs to be conducted on-the-fly during the state space exploration.

To apply the sweep-line method a *progress measure* must be provided for the model as formalised below where $\mathcal{S}$ denotes the set of all states (markings) and $\rightarrow^*$ the reachability relation of the CPN model:

Definition 6.1 (Progress Measure)**.** A **progress measure** is a tuple $\mathcal{P} = (O, \sqsubseteq, \psi)$ such that O is a set of **progress values**, $\sqsubseteq$ is a total order on O, and $\psi : \mathcal{S} \rightarrow O$ is a **progress mapping**. $\mathcal{P}$ is **monotonic** if $\forall s, s' \in \mathcal{R}(\iota) : s \rightarrow^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$. Otherwise, P is **non-monotonic**. □

The sweep-line method does not mandate any origin of the progress measure and in many cases the progress measure is provided by the modeller based upon knowledge about the modelled systems. For PA-CPNs, however, a reasonable progress measure can be derived from the model automatically as we will show in the next section.

6.2 Progress Measures for Sweep-line Verification Methods

For our example protocol above, the progress measure could be a vector of measures using the number of tokens on some of its places (omitting the parts of the model that we did not show in this paper):

$$(|d0|, |d1|, |d2|, |d3|, |startSending| + |next|, |end|, \ldots)$$

The order on two such vectors would be compared lexicographically, meaning the order of the different entries represents their significance.

The first four entries represent the progress in the service tester (Fig. 11). The next two entries represent the progress within the send service (Fig.4); note that since the places *startSending* and *next* are on a loop, tokens can flow back from place *next* to place *startSending*. The *end* place is actually the respective driver place from the tester, which propagates the progress between the service and tester. Therefore, the tokens on both places within this loop are counted the same (added up in the same entry of the vector).

An alternative progress measure is shown below (omitting the parts of the model that we did not show in this paper):

$$(|d0|, |d1|, |d2|, |d3|, |startSending|, |next|, |end|, \ldots)$$

The difference between the two are based on how loops are handled. In this progress the places on loops are append to the vector as if the loop was not there. In the present example this is shown by havin replaced the + operator between *startSending* and *next* with a comma.

We used the service tester as the first and, therefore, most significant measures since these are indicating the progress within the test. Then we measure the progress within the service. In some cases, it might make sense to take life-cycle variables into account for measuring the progress. But, this depends very much on the protocol and whether the life-cycle variables monotonically increase in the course of the protocol. In our example, the life-cycle variable *ready* of the sender module does not indicate any progress; therefore, it is not part of the progress measure that we have shown above.

Generally, coming up with a good progress measure requires some experience and a good understanding of the protocol. For the test drivers and the services modules, however, some reasonable progress measures can be derived automatically by exploiting the block structure of the respective modules (a sequence for the tester and a loop for the service, in our example). We formalize this below, basically generalizing the idea from the above example.

The progress measure is defined on top of the tree decomposition of the blocks underlying the corresponding service tester model or the service module. Technically, the tree decomposition of the blocks was formally defined for service level modules only. It is straight forward to adjust this definition to service tester modules, but we do not formalize that here. In the following, we assume that we have the tree decomposition of the respective module. Then we define a simple progress measure and a complex one. The simple one, would just add up the number of all tokens in loops, not looking into their detailed structure; the complex one would also take the progress within loops into account.

Definition 6.2 (Progress measures)**.** Let BT be a block tree for a CPN module.

The sequence of *simple progress* measures entries for BT is defined inductively over the block tree BT of the CPN module:

- If BT is $B : atomic$, then simple progress sequence consist of $|s|, |e|$ where s is the entry place of the block B and e is the exit place.
- If BT is $B : sequence$ with sub blocks $B_1, \ldots B_n$, and $e_i^1, \ldots, e_i^{k_i}$ are the simple progress sequences for B_i, then $e_1^1, \ldots, e_1^{k_1-1}, e_2^1, \ldots, e_2^{k_2-1}, \ldots, e_n^1, \ldots, e_n^{k_n}$ is the simple progress sequence for BT.
- If BT is $B : choice$ with sub blocks $B_1, \ldots B_n$, and $e_i^1, \ldots, e_1^{k_i}$ are the simple progress sequence for each block B_i, then the sequence $e_1^1, \ldots, e_1^{k_1-1}, e_2^2, \ldots, e_2^{k_2-1}, \ldots, e_n^2, \ldots, e_n^{k_n}$ is the simple progress sequence for BT
- If BT is $B : loop$ with places $p_1, \ldots p_n$, then either the single entry $|p_1| + \ldots + |p_n|$ is the simple progress sequence for BT.

The sequence of *complex progress* measures entries is defined inductively over the block tree BT of the CPN module:

- If BT is $B : atomic$, then complex progress sequence consist of $|s|, |e|$ where s is the entry place of the block B and e is the exit place.

- If BT is B : *sequence* with sub blocks $B_1, \ldots B_n$, and $e_i^1, \ldots, e_i^{k_i}$ are the complex progress sequences for B_i, then $e_1^1, \ldots, e_1^{k_1-1}, e_2^1, \ldots, e_2^{k_2-1}, \ldots, e_n^1, \ldots, e_n^{k_n}$ is the simple sequence for BT.
- If BT is B : *choice* with sub blocks $B_1, \ldots B_n$, and $e_i^1, \ldots, e_1^{k_i}$ are the complex progress sequence for each block B_i, then the sequence $e_1^1, \ldots, e_1^{k_1-1}, e_2^2, \ldots, e_2^{k_2-1}, \ldots, e_n^2, \ldots, e_n^{k_n}$ is the complex progress sequence for BT
- If BT is B : *loop* with places with sub block B_1 and B_2 with the complex progress sequence $e^1, \ldots, e^n$ for B_1, then $e^1, \ldots, e^n$ is also the complex progress sequence for BT.

Now, the progress measures for the complete system can be built from the progress sequences (either the simple or the complex ones) for the tester and service modules by concatenating the sequences: The concatenation would first choose the sequences for the service testers and then the sequences for all the service level modules. Note that if there is a driver place of a service tester attached to the service, this driver place would also be added to the progress measure sequence of the service level module at the end (as for the *end* place for the send service in our example).

Note that for each single service either the simple or the complex measure could be chosen. It very much depends on the nature and the depth of the loop which of these choices help reducing the effort for exploring the state space. The choice for each service level module would be left to the modeller. In principle, it is even possible to combine the complex measure and the simple measure within a single service level module – starting with the complex measure for the outer loop constructs and then switching to the simple measure at some nesting level. But, defining these combined measures for a single service level module would result in a very technical definition. Therefore, we do not formally define combined complex and simple measures for a single service here.

Anyway, as for all heuristics, these measures serve as a good guess only; it might still be possible to be improved by choosing the simple or the complex progress measures for the different modules or by by adding some entries concerning the the live-cycle variables. Such manual manipulations would also be left to the modeller.

6.3 Verification Results

In order to demonstrate the feasibility of verification using the sweep-line method and the progress measures defined above, we present the verification of a simple end state property assuring that the protocol will terminate in a consistent state. The property checks that all the modules are ended in all final states. This was checked manually and by automatically checking all end states with the simple predicate *P1* shown below. The predicate says that the four places d3, endFinalAtomic,endRecAck and endRec are all marked with a unit token. This means that the service tester and all the services are terminated.

- **P1:** d3 = [()] andalso endFinalAtomic = [()] andalso endRecAck = [()] andalso endRec = [()]

Configuration	Visited states	Peak stored	Num unique end states	P1	time
1 msg, non-lossy	156	77	2	yes	0.034905s
1 msg, lossy	186	99	2	yes	0.029867s
3 msg, non-lossy	2222	2014	4	yes	0.399659s
3 msg, lossy	2928	2700	4	yes	0.643134s
7 msg, non-lossy	117584	115373	8	yes	216.197694s
7 msg, lossy	160620	158388	8	yes	532.674399s

Table 1: Verification results using the simple progress measure

Configuration	Visited states	Peak stored	Num unique end states	P1	time
1 msg, non-lossy	165	63	2	yes	0.029353s
1 msg, lossy	196	78	2	yes	0.035034s
3 msg, non-lossy	2790	1582	4	yes	0.489735s
3 msg, lossy	4037	2187	4	yes	0.855804s
7 msg, non-lossy	143531	86636	8	yes	32.384360s
7 msg, lossy	263608	124661	8	yes	80.835973s

Table 2: Verification results using the complex progress measure

We ran the verification using two different progress measures. The results of the verification are shown in Table 1 using the simple progress measure and Table 2 using the complex progress measure. We used two configuration parameters, the number of messages to be sent and whether the channel is lossy. We see that the predicate holds for all configurations. Also, we see that the number of states grows fairly fast with the number of messages. We see that the verification using the simple progress measure consistently visits fewer states (counting duplicate encounters twice) than the complex progress measure. This is because the states surrounding and inside loops are added, which means they count as a single progress measure point by the simple progress measure, while they are all included as if the loop was a sequence using the complex progress measure. The peak number of states stored at the same time, however, is consistently lower for the complex progress measure. This means that the peak memory consumption, is lower using this metric. Furthermore, for the larger state spaces, the time the verification takes is also significantly lower using the complex progress measure. This means that the complex measure is probably preferable with large state spaces when the model includes loops. It is likely that a tailored progress measure would out-perform both the complex and simple progress measure.

7 Conclusion and Related Work

In this paper we have presented a formal definition of Pragmatics Annotated Coloured Petri Nets (PA-CPNs), the net class that forms the basis for our code generation technique. Furthermore we have shown that the structure of PA-CPNs can be exploited to automatically derive some suitable progress measures for the sweep-line method.

PA-CPNs are not the first formally defined sub-class of CPNs for code generation: also Process-Partitioned CPNs (PP-CPNs) [4, 7] were defined for making code generation possible. One important advantage of PA-CPNs over PP-CPNs

is that they clearly display the available services of a principal in the PLMs. With PA-CPNs, we are also able to use the PLMs to reduce the number of states in memory at any one time during state-space generation by taking into account the LCV places, even though we could not exploit that in the example discussed in this paper.

PA-CPNs have been introduced mainly with code generation in mind (with different objectives as discussed in [5]). In this paper, we have formally defined PA-CPNs, and it turned out that the objective of code generation does not spoil the possibility of verifying the respective models; on the contrary, the additional structure can even be exploited for improving verification in combination with the sweep-line method.

References

[1] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS, Proceedings*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.

[2] K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[3] K. Jensen, L.M. Kristensen, and T. Mailund. The sweep-line state space exploration method. *Theoretical Computer Science*, 429:169–179, 2012.

[4] L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.

[5] K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.

[6] K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. In *To appear in Proc. of WS-FMDS 2013*, 2013.

[7] M. Westergaard. Verifying parallel algorithms and programs using coloured petri nets. In *TOPNOC VI*, volume 7400 of *LNCS*, pages 146–168. Springer, 2012.

Chapter 11

PetriCode: A Tool for Template-based Code Generation from CPN Models

PetriCode: A Tool for Template-based Code Generation from CPN Models

Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computing, Bergen University College, Norway
Email: {kifs}@hib.no
[2] DTU Compute, Technical University of Denmark, Denmark
Email: {kisi}@imm.dtu.dk

Abstract. Code generation is an important part of model driven methodologies. In this paper, we present PetriCode, a software tool for generating protocol software from a subclass of Coloured Petri Nets (CPNs). The CPN subclass is comprised of hierarchical CPN models describing a protocol system at different levels of abstraction. The elements of the models are annotated with code generation pragmatics enabling PetriCode to use a template-based approach to generate code while keeping the models uncluttered from implementation artefacts. PetriCode is the realization of our code generation approach which has been described in previous works.
Keywords: Model-driven development, Implementation of platforms and tools, Formal methods for software engineering, Coloured Petri Nets.

1 Introduction

Coloured Petri Nets (CPNs) [5] is a graphical modelling language combining Petri Nets and the programming language Standard ML. CPNs have been widely used for modelling and validation of concurrent systems. CPN Tools [6] provides tool support for construction, simulation and analysis of CPN models but does not provide tool support for automatic code generation from CPN models. The contribution of this paper is to present PetriCode which complements CPN Tools by providing tool support for automatic code generation from CPN models. PetriCode implements the approach presented in [19].

In contrast to previous works [16, 18, 19], this paper focuses on the technical software realization of our approach whereas earlier work has focused on the conceptual and theoretical aspects of our modelling and code generation methods. The intended use of PetriCode is to generate software for network protocols in a flexible way based on annotated and descriptive protocol CPN models [18] and for different target languages and platforms.

PetriCode takes a template-based approach to code generation based on CPN models annotated with *pragmatics*. Pragmatics are syntactic annotations on CPN model elements that are used to direct the code generation procedure. Pragmatics are associated with code templates that are invoked for code generation. Our code generation approach [19] consists of three main steps. The first

step is to parse the CPN model and automatically derive additional pragmatics for the CPN model. The derived pragmatics are used to provide the code generator with additional information of what is represented by the various CPN structures. The second step is to construct an Abstract Template Tree (ATT) which is used as an intermediary structure for code generation. The ATT provides a platform independent data structure that simplifies the final step of the code generation. The third and final step is the actual code generation where the ATT, using a series of visitors and templates, is transformed into code by invoking the templates associated with pragmatics.

The rest of this paper is organized as follows. Section 2 shows, by an example, how PetriCode can be used to generate code for a simple framing protocol. Section 3 provides an overview of the software architecture and design of PetriCode. Section 4 describes the pragmatics module which is responsible for parsing and deriving pragmatics. Section 5 describes the ATT module which is responsible for generating the ATTs. Section 6 describes the code generation module which is responsible for generating code based on ATTs and templates. Section 7 contains a discussion of related work. Concluding remarks and future work are presented in Sect. 8.

We assume that the reader is familiar with the basic concepts of Petri Nets (places, transitions, enabling and occurrence/firing). Due to space limitations we only provide a high-level introduction to CPNs. The reader is referred to [5] for a detailed introduction to CPNs.

Details on how to download and operate PetriCode are available at the PetriCode project website [15]. Due to space limitations we cannot present all details of PetriCode in this paper. For a more detailed presentation, we refer the reader to the technical report [17].

2 Example Model and Usage

In order to present the workings of PetriCode, we use a simple framing protocol as a running example. The protocol is described in detail in the technical report [19]. The model is divided into three hierarchical layers: the protocol system, principal, and service layers. The protocol system layer, depicted in Fig. 1, shows the principal agents of the protocol system as well as the connections between them. In the example, those are the Sender, Receiver and the Channel connecting them. In Fig. 1, the substitution transitions Sender and Receiver (rectangles with double-lined borders) are both annotated with a $\langle\langle$principal$\rangle\rangle$ pragmatic. This conveys to the code generator that the sub-modules represented by each of these substitution transitions represent principal agents of the system. The third substitution transition in the protocol system module, Channel, is annotated with the pragmatic $\langle\langle$channel$\rangle\rangle$ specifying that the underlying module defines the channel. The $\langle\langle$channel$\rangle\rangle$ pragmatic, in addition, has some attributes describing the service provided by the channel. In the rest of this paper, we focus on the Sender principal of the protocol. Figure 2 shows the principal level of the sender which is the sub-module of the substitution transition Sender in Fig. 1.

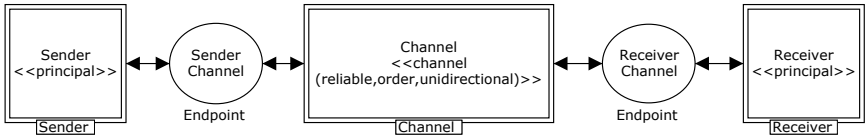


Fig. 1. The protocol system level

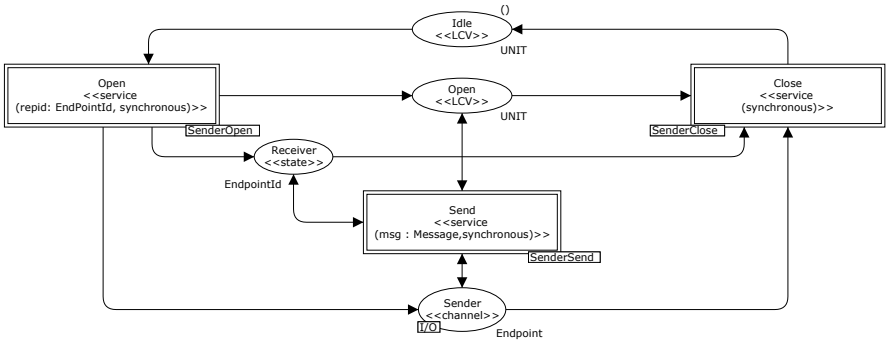
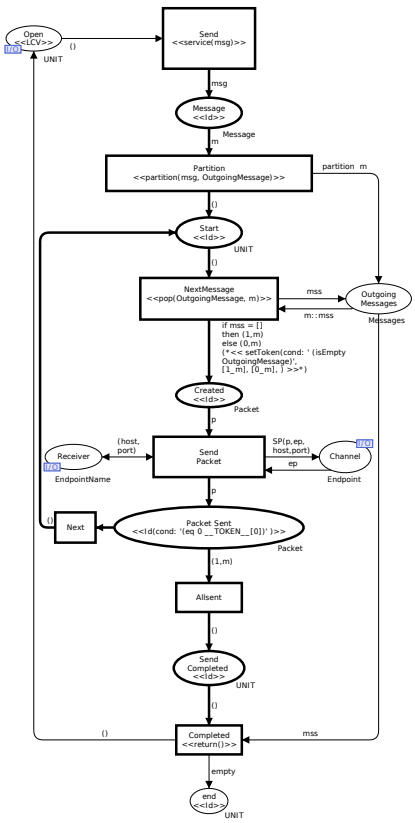


Fig. 2. Example of a principal level module: The Sender module

The principal level contains the services provided by each principal as well as *life cycle variables* which control when the various services can be called and places which hold global data for the principal. The Open and Close services (represented by substitution transitions with a ⟨⟨service⟩⟩ pragmatic) opens and, respectively, closes the channel to the Receiver while the Send service sends a message over the channel. To illustrate the models of the services, we provide details on the Send service. The Send service, shown in Fig. 3(left), contains the sending part of the protocol. The Send service divides a message into smaller fragments called frames. Each frame is sent together with a bit (flag) that is set if the current frame is the last frame of the message, and unset otherwise. In the model, the message, which is a parameter to the Send service, is broken up into frames by the transition Partition. Then the fragments are sent one by one in a loop (from the Start place to the PacketSent place) until all the fragments have been sent. The ⟨⟨service⟩⟩ pragmatic is used on transition Send (top) to indicate the entry point of the service. At the bottom of Fig. 3, the pragmatic ⟨⟨return⟩⟩ on the Completed transition indicates the termination of the service.

Usage Example. In order to generate code from the CPN model, PetriCode is invoked with appropriate arguments. An example of such an invocation is shown in Listing 1. The first step of the program is to parse the model and automatically add *derived pragmatics*. It is also possible, as part of the command-line arguments, to give further pragmatics and rules for deriving them as will be discussed in Sect. 4. The second step is to generate the ATT which is discussed




```
1
2   def Send(msg){
3     if(!Open)
4       throw new Exception(
5       'unfulfilled '+
6       'precondition: Open')
7     Open = false
8
9
10    def __TOKEN__
11    def OutgoingMessage
12    def __LOOP_VAR__
13
14    OutgoingMessage = msg
15      .getChars().toList()
16      .collate(5).collect{
17        new String(it.toArray(
18          new char[0]))
19      }
20
21    __LOOP_VAR__ = true
22    while(__LOOP_VAR__){
23      /*vars: [m]*/
24      def m
25      m = OutgoingMessage
26            .remove(0)
27      if(OutgoingMessage.size()
28          == 0){
29        __TOKEN__ = [1,m]
30      } else {
31        __TOKEN__ = [0,m]
32      }
33      Receiver
34        .getOutputStream()
35        .newObjectOutputStream()
36        .writeObject( __TOKEN__)
37      __LOOP_VAR__ =
38        ( 0 == __TOKEN__[0] )
39    }
40    Open = true
41    return
42  }
```

Fig. 3. The Sender Send module(left) and generated code (right)

further in Sect. 5. The third and final phase is the code generation where the `-o` option provides the output directory where the generated code is placed and the `-b` option takes a *binding descriptor* file as an argument. The binding descriptor file provides a set of bindings of pragmatics to code generation templates for the specific platform under consideration. These bindings (known as *template bindings*) are described in further detail in Sect. 6. In this case binding descriptors for the Groovy platform are used. One thing that is not visible in the listing is a reference to pragmatics descriptors which describes the available pragmatics. This is because a core set of pragmatics, which contains most of the pragmatics used in this particular example, are defined in the tool and available by default.

Listing 1. Command to run PetriCode for the simple framing protocol example.

```
petriCode -o . -b ./groovy.bindings ./FramingProtocol.cpn
```

After running the command shown in Listing 1, two files will be generated in the output directory. Each of these files contain a single Groovy class, one for the Sender principal and one for the Receiver principal. For the Sender class there will be exactly three methods, one for each of the services that the principal provides (see Fig. 2). The generated code for the Send service is shown in Fig. 3 (right).

3 Architecture and Design of PetriCode

PetriCode is divided into three functional modules corresponding to the three main steps in our code generation approach. These are the Pragmatics, ATT, and Code generation modules.

When designing and implementing PetriCode, there was a number of key requirements that needed to be addressed and which affected the choice of software technologies used for the implementation. An important feature of PetriCode is the ability to read, parse and write CPN models stored in the format of CPN Tools [6] which is one of the most widely used tools for construction and analysis of high-level Petri Nets. The Java library Access/CPN [21] provides this capability for the Java platform. Therefore, in order to use Access/CPN it is necessary to choose a platform with good integration with Java. Furthermore, in order to accommodate pragmatics it is required to be able to refine the meta-model underlying Access/CPN without introducing a complicated translation layer. Another important requirement was to easily be able to create Domain Specific Languages (DSLs) for defining pragmatics descriptors and template bindings. The Groovy programming language [3], which runs on the Java Virtual Machine, was chosen since it has a seamless integration with all Java libraries including Access/CPN. Groovy also has a simple mechanism (not available to Java) to manipulate classes at runtime and also has good support for many types of DSLs. Finally, Groovy has additional useful features such as a command-line interface options builder and a powerful template engine that can be used for code generation purposes.

Overall Architecture. Figure 4 provides an architectural overview of PetriCode. PetriCode is controlled by its main class PetriCode which makes up the Command Line Interface of the application. PetriCode parses the command-line arguments and calls the modules shown directly below the Command Line Interface in Fig. 4 as appropriate. PetriCode uses the CliBuilder included in Groovy to parse command line arguments. All the modules depend on Access/CPN for reading and manipulating CPN models. As explained above, PetriCode is implemented using the Groovy language and builds upon the Groovy and Java platforms. All modules are dependent on the data model for Pragmatics. The ATT and Generation modules also share a data model for ATTs.

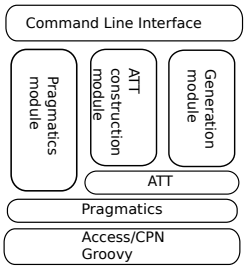


Fig. 4. PetriCode Architecture

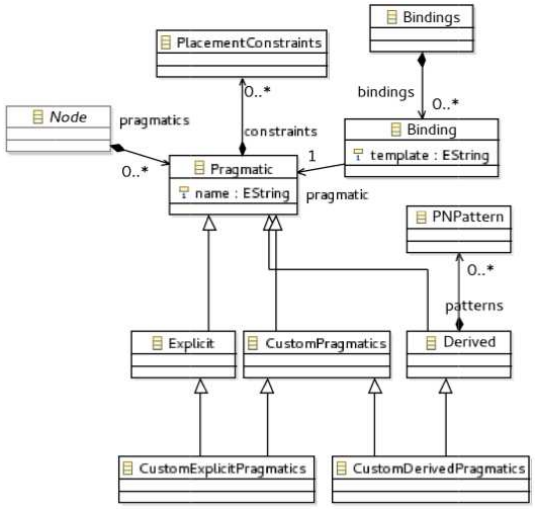


Fig. 5. Data model for the Pragmatics module

4 Pragmatics Module

The Pragmatics module has three main responsibilities: reading and parsing CPN models, parsing *pragmatics descriptors*, and computing derived pragmatics for CPN models. The pragmatics derivation process is driven by a DSL which is used to parse the pragmatics descriptor files containing information about the pragmatics used in a model. A class diagram showing the meta-model for pragmatics is provided in Fig. 5. In the diagram, pragmatics are separated via two categorizations. One categorization is whether the pragmatic is explicit or derived, where explicit pragmatics must be added to the CPN model by the modeller, and derived pragmatics are computed automatically based on structural patterns. This categorization is represented in Fig. 5 by the Derived class. The second categorization is whether the pragmatic is supplied by the user (a custom pragmatic) or is part of the built-in core pragmatics of PetriCode. This is represented by the CustomPragmatics class.

The pragmatics description language is a builder language that describes the available pragmatics. Listing 2 gives an example of a pragmatic descriptor for an explicit pragmatic (⟨⟨principal⟩⟩) and a derived pragmatic (⟨⟨endLoop⟩⟩). A core set of pragmatics is provided by PetriCode while others can be provided by the user using the pragmatics description language. The language consists of *descriptors* that each describe a pragmatic. Each descriptor consists of a name (which is the name of the pragmatic) followed by a pair of parenthesis. Inside the parenthesis, the parameters of the pragmatics definition are given in the form of key-value pairs. The possible parameters for a pragmatics descriptor are `origin` and `derivationRules`. The `origin` parameter indicates whether the prag-

matic is explicitly given by the modeller or should be automatically derived. The origin field of ⟨⟨Principal⟩⟩ indicates that this is an explicit pragmatic meaning that it will not be generated automatically. The `derivationRules` parameter gives structural patterns that is used to find the elements of a CPN model where a derived pragmatic should be added. In addition, both ⟨⟨Principal⟩⟩ and ⟨⟨endLoop⟩⟩ have some constraints on where they may reside in the model which is supplied via the `constraints` field.

Listing 2. Examples of the core pragmatics for PetriCode

```
principal(origin: 'explicit', constraints: [levels: 'protocol',
          connectedTypes: 'SubstitutionTransition'])
endLoop(origin: 'derived', derviationRules:
        ['new PNPattern(pragmatics: [\'Id\'],
          minOutEdges: 2, backLinks: 1)'],
        constraints: [levels: 'service', connectedTypes:'Place'])
```

Pragmatics Derivation. The method for deriving pragmatics is based on traversing each service module and checking each node (i.e, place or transition) against structural patterns described by the pragmatic descriptors. The last pragmatic descriptor in Listing 2 is the ⟨⟨endLoop⟩⟩ pragmatic. ⟨⟨endLoop⟩⟩ is a derived pragmatic with a structural pattern on the field *derivationRules*. An important concept for pragmatics derivation and indeed the entire code generation approach is the *control flow path*. The control flow path consists of all the nodes annotated with the ⟨⟨Id⟩⟩ pragmatic where the first node would be the node of a service annotated with ⟨⟨service⟩⟩ pragmatic and the last is annotated with ⟨⟨return⟩⟩ (see Fig. 3). Each of the ⟨⟨Id⟩⟩, ⟨⟨service⟩⟩ and ⟨⟨return⟩⟩ pragmatics are explicit and must be added by the modeller. For derived pragmatics, a list of patterns are supplied. Each pattern, will be matched against each node on the control-flow path. If a pattern matches, the corresponding pragmatic is added to the node.

5 ATT Construction Module

The ATT module is responsible for generating ATTs and the main classes that make up the ATTs are shown in Figure 6. An ATT is an internal temporary data structure of PetriCode. Its purpose is to simplify the code generation process and make it more flexible by organizing so-called control flow blocks at the service modules in an ordered tree. When this tree has been constructed, code generation is performed by traversing the tree. The tree is built up according to the hierarchical structure of the considered subclass of CPN models down to the service level. At the service level, the control flow structure of the service is reflected in the structure of the ATT.

The ATT generation is done by the ATTFactory class which produces an instance of the class AbstractTemplateTree. The AbstractTemplateTree has as its descendants instances of the classes Atomic, Conditional, Loop, Principal and Service corresponding to the different kinds of control flow blocks. The Principal and

Service classes each have a link going to the Instance class of the Access/CPN model which represents substitution transitions. The Block class has two outgoing associations with Place nodes from Access/CPN. The Atomic block has an association with transitions.

An ATT is implemented as an ordered tree. Each non-leaf element in the tree has a list of children. The root element of an ATT is an instance of the AbstractTemplateTree class. Each child of the root element is expected to be of the class Principal. The Principal class has as its children the services of the principal. The Service class represents a service, its children are the control flow blocks of the service according to the block structure introduced in [19].

The ATT of the Sender side of the example in Sect. 2 is shown in Fig. 7. The tree has a single root representing the entire protocol system. At the next level, the principals are represented. For brevity, only the principal Sender of the protocol is shown. The children of the principal nodes are the services, and their children represent the control flow block structure of the services. Looking specifically at the Send service of the Sender principal, we see that the service has three direct descendants. These descendants represent the loop in the service and one atomic block on each side of the loop. The first of the nodes is the partition atomic block which contains the partition pragmatic which is where the message sent by the framing protocol is divided into smaller fragments. The second node is the loop, and the final node is the atomic block after the loop which does not have any pragmatics and as such does not produce any code.

6 Code Generation Module

The generation module is responsible for generating code from ATTs. In order to generate code from CPNs annotated with pragmatics, the pragmatics must be connected to code generation templates. This is done using the Binding class which is connected to Pragmatics (see Fig. 5). The bindings are produced by another DSL which parses user provided template bindings and returns an object structure for the template bindings. The code generation phase can be divided into two separate sub-phases. The first sub-phase is the code generation for each element in the ATT. A visitor visits each element in the ATT in no particular order. The second sub-phase in the code generation phase is to stitch together the generated code for each ATT node. This is done by a depth-first traversal of the ATT. For each node, when all the sub-nodes have been visited, the `%%yield%%` tag in the code generated for the node is replaced by the concatenation of the text field of all the immediate descendants of the node. When this has been done for each principal in the protocol, the code generation is complete and the code is written to the output directory.

Template bindings. In order to select the proper code template for each pragmatic, the user supplies PetriCode with *template bindings*. These bindings are supplied using a DSL. The DSL allows the user to specify the template and other necessary information about a template and how it should be applied.

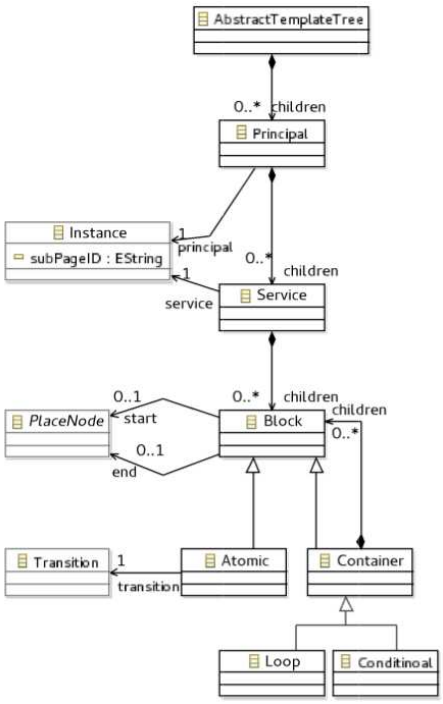


Fig. 6. Classes of the ATT

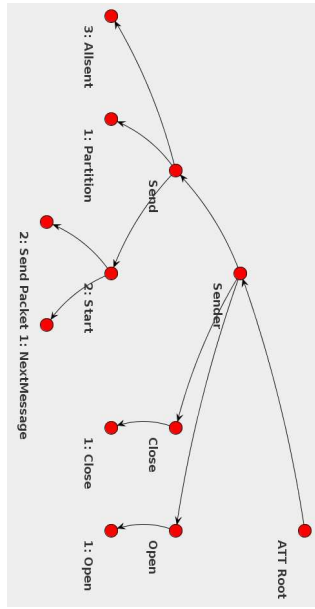


Fig. 7. Example ATT

Listing 3 shows two examples of template bindings. The first binding is a binding for the ⟨⟨Principal⟩⟩ pragmatic, which is used on the Sender and Receiver substitution transitions in Fig. 1. This is a container, which means that the generator should add the code generated to the principals children in the ATT to it. The other fields are `pragmatic` (which names the pragmatic) and `template` (which contains the file-name of the template). The second template binding binds ⟨⟨endLoop⟩⟩, which is placed on the Completed place (see Fig. 3) after pragmatics derivation. In addition, it is possible to add the field parameterStrategy to template bindings. This field determines how the parameters of the template should be constructed.

Listing 3. Two examples of template bindings.

```
classTemplate(pragmatic: 'Principal',
    template: './groovy/mainClass.tmpl', isContainer: true)
endLoop(pragmatic: 'endLoop',
    template: './groovy/endLoop.tmpl')
```

7 Related Work

Many tools exist for generating software from models. Most of the tools, however, support only the generation of static parts of the code and, partly, standard behaviour [7]. This does less than it could to help create robust software since the non-trivial parts are still written manually. However, some tools allow for generating more than structural parts of software. In the discussion on related work below, we consider only tools and approaches that do full code generation where no manual coding is necessary.

Process-Partitioned CPNs (PP-CPNs) [9] have been used to automatically generate code for several purposes including protocol software. PP-CPNs are a restricted sub-class of CPNs. Code is generated from PP-CPNs by first translating the PP-CPN into a control flow graph (CFG), then translating the CFG into an abstract syntax tree for an intermediate language. The CFG is translated into another intermediary representation which is dependent on the target platform, and from this representation code is generated. In [9], PP-CPNs are used to model and obtain an implementation for the DYMO routing protocol using the Erlang programming language and platform. Both PP-CPNs and our modelling language are subclasses of CPNs. However, where we rely on pragmatics to control code generations, PP-CPNs rely on restricted colour sets and CPN structure to allow the generator to deduce the needed information. Our approach also models the environment of the services while PP-CPNs are geared to modelling only the intents of the services. This allows us to represent the protocol at higher levels of abstraction on the protocol and principal levels as well as on the service level. It also allows us to define how the services should be called in a structured way by third-party software.

There are several tools for modelling and generating protocol software based on the Specification and Description Language (SDL) [2, 4]. SDL is created for the purpose of modelling protocols, and is extensively used in the telecommunications industry. The IBM Rational SDL Suite (previously Tau SDL Suite and SDT) is among the most well known proprietary tools for SDL. The Rational SDL Suite supports code generation for SDL models to C and C++ code and also supports verification through model checking. Another SDL tool is Jade [14] that supports editing and analysis/verification of SDL models. Code generation for JADE is still in development. SDL Integrated Tool Environment (SITE) supports editing of SDL models and code generation to Java and C++ code. SITE also supports some analysis of SDL models. SDL is a graphical language based on Finite State Machines (FSMs). This allows verification of protocols using model checking techniques. Compared to our approach, SDL is not as easily extensible as our approach.

Renew [12] is a tool that allows creation and execution of object-oriented Petri Nets. Renew supports several modelling formalisms based on various forms of Petri Nets. Renew supports Reference nets which can be annotated with Java code and can be executed using a built-in simulator engine. The simulator can execute the nets incorporating the Java annotations in a headless mode so that no visualization will occur. This means that the simulations can be used

as stand-alone programs. The simulation approach is in contrast to our code generation approach where code is generated and can be inspected and compiled as computer programs created with traditional programming languages.

The Unified Modelling Language, and in particular state charts and sequence diagrams, has been used to model and generate code for protocols in several approaches [1, 10, 11, 13, 20]. Several tools exists for UML which support analysis and code generation in various ways. Since our approach is based on CPNs, verification is directly supported using CPN Tools [8]. This may be more challenging with UML-based approaches. Also, our pragmatics- and template-based approach allows us to give the user a great deal of flexibility by supporting the definition of custom pragmatics and templates.

8 Conclusions and Future Work

In this paper we have described a tool that can generate code from CPNs annotated with pragmatics. We have shown how this tool works by using the example of a simple communication protocol. The goal of our tool is to be able to generate code that is complete in the sense that no further coding should be required to use the services our code provides. Another important goal has been to generate code that is readable and analysable for human programmers.

The input of the tool is an instance of a specific class of CPN models. A main goal of the tool, and of our approach in general, is that these models should be descriptive in the sense that they can be used to convey the operation of the modelled protocol at several levels of abstraction.

In the future, we will use the tool to evaluate our approach using a larger and more realistic examples, and expand the range of available templates to other languages and platforms. Another future work item we are currently working on is to make our approach more flexible by allowing the users to easily add custom pragmatic patterns and placement conditions. Finally, we aim at integrating PetriCode with other popular software development tools such as Eclipse and IntelliJ IDEA.

References

1. M. Alanen, J. Lilius, I. Porres, and D. Truscan. *On Modeling Techniques for Supporting Model Driven Development of Protocol Processing Applications*, pages 305–328. Springer, 2005.
2. F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *Communications Surveys Tutorials, IEEE*, 4(1):2–20, 2002.
3. Groovy. *Project Web Site*. http://groovy.codehaus.org.
4. ITU-T. Recommendation Z.100 (11/99) Specification and Description Language (SDL), 1999.
5. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

6. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.
7. E. Kindler. Model-based Software Engineering; The challenges of modelling behaviour. In *Proc. of BM-FA '10*, pages 4:1–4:8. ACM electronic libraries, 2010.
8. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Transactions on Petri Nets and Other Models of Concurrency VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
9. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
10. C. Kroiss, N Koch, and A. Knapp. UWE4JSF: A Model-Driven Generation Approach for Web Applications. In *Proc. of ICWE '09*, volume 5648 of *LNCS*, pages 493–496. Springer, 2009.
11. P. Kukkala, V. Helminen, M. Hannikainen, and T.D. Hamalainen. UML 2.0 implementation of an embedded WLAN protocol. In *Proc. of PIMRC '04*, volume 2, pages 1158–1162 Vol.2, 2004.
12. O. Kummer, F. Wienberg, M. Duvigneau, M. Schumacher, J.and Köhler, D. Moldt, H. Rölke, and R. Valk. An extensible editor and simulation engine for petri nets: Renew. In *Proc. of ICATPN '04*, volume 3099 of *LNCS*, pages 484–493. Springer, 2004.
13. J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. UML for protocol engineering-extensions and experiences. In *Proc. of TOOLS '00*, pages 82–93, 2000.
14. C.L. Pereira, Jr. da Silva, D.C., R.G. Duarte, A.O. Fernandes, L.H. Canaan, C.J.N. Coelho, and L.L. Ambrosio. Jade: An embedded systems specification, code generation and optimization tool. In *Proc. SBCCI '00*, pages 263–268, 2000.
15. PetriCode. *Project Web Site*. http://kentis.github.io/petriCode/.
16. K.I.F. Simonsen. On the use of Pragmatics for Model-based Development of Protocol Software. In *Proc of PNSE '11*, volume 723 of *CEUR Workshop Proceedings*, pages 179–190. CEUR-WS.org, 2011.
17. K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. Technical Report DTU Compute-Technical Reports-2013-11, PetriCode, 2013.
18. K.I.F. Simonsen and L.M. Kristensen. Towards a CPN-based Modelling Approach for Reconciling Verification and Implementation of Protocol Models. In *Proc. of MOMPES'12*, volume 7706 of *LNCS*, pages 106–125. Springer, 2012.
19. K.I.F Simonsen, L.M. Kristensen, and E. Kindler. Code Generation for Protocol Software from CPN models Annotated with Pragmatics. In *To appear in proc. of SBMF'13*, LNCS. Springer.
20. M.A. Wehrmeister, E.P. Freitas, C.E. Pereira, and F. Rammig. Genertica: A tool for code generation and aspects weaving. In *Proc. of ISORC '08*, pages 234–238, Washington, DC, USA, 2008. IEEE Computer Society.
21. M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In *Proc. of ICATPN '09*, volume 5606 of *LNCS*, pages 313–322. Springer, 2009.

Chapter 12

An Evaluation of Automated Code Generation with the PetriCode Approach

An Evaluation of Automated Code Generation with the PetriCode Approach

Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computing, Bergen University College, Norway
Email: kifs@hib.no
[2] DTU Compute, Technical University of Denmark, Denmark

Abstract. Automated code generation is an important element of model driven development methodologies. We have previously proposed an approach for code generation based on Coloured Petri Net models annotated with textual pragmatics for the network protocol domain. In this paper, we present and evaluate three important properties of our approach: platform independence, code integratability, and code readability. The evaluation shows that our approach can generate code for a wide range of platforms which is integratable and readable.

1 Introduction

Coloured Petri Nets (CPNs) [5] is a general purpose formal modelling language for concurrent systems based on Petri Nets and the Standard ML programming language. CPNs and CPN Tools have been widely used to model and validate network protocol models [6]. In previous works [14], we have proposed an approach to automatically generate network protocol implementations based on a subclass of CPN models. We have implemented the approach in the PetriCode tool [13]. In this approach, CPN models are annotated with syntactical annotations called *pragmatics* that guide the code generation process and have no other impact on the CPN model. Code is then generated based on the pragmatics and code generation templates that are bound to each pragmatic through template bindings. This paper presents an evaluation of the PetriCode code generation approach and tool.

The four main objectives of our approach are: platform independence, code integratability, code readability, and verifiability of the CPN models. The contribution of this paper is an evaluation of the first three of these objectives. In this study, we used the PetriCode [13] tool to evaluate our code generation approach. Platform independence, i.e., the ability to generate code for several platforms, is an important feature of our approach. For the purposes of this study, a platform is a programming language and adjoining APIs. Being able to generate protocol implementations for several platforms allows us to automatically obtain implementations for many platforms based on the same underlying CPN model. Platform independence also contributes to making sure that implementations for different platforms are interoperable. Another aspect is to have

models that are independent of platform specific details. Integrateability, i.e., the ability to integrate generated code with third-party code, is important since the protocols must be used by other software components written for the platform under consideration (upwards integratability). It is also important to be able to support different underlying libraries so that the generated code can be referred to by other components (downwards integrateability). Readability is important in order to gain confidence that the implementation of a protocol is as expected. While being able to verify the formal protocol models also contribute to this, inspecting and reviewing the final code further strengthens confidence in the correctness of the implementations. The ability to manually inspect the generated code is useful since, in our approach, we only verify the model which is not sufficient to remove local errors in the code.

The rest of this paper is organized as follows. Section 2 describes the example protocol used throughout this paper, and illustrates the code generation process for the Groovy platform. Section 3 evaluates platform independence by considering the Java, Clojure, and Python platforms. Section 4 evaluates integrateability, and Section 5 evaluates readability of the code generated by our approach. Section 6 presents related work, sums up conclusions and discusses directions for future work. Due to space limitation we provide little on CPNs and Petri Nets. The reader is referred to [5] for details on CPNs and Petri Nets. The PetriCode tool as well as the model, template and bindings used in this paper are available at [10].

2 Example and Code Generation

In this section, we present an example CPN model which is an extension of the protocol model we have used in previous work [14]. The example allows us to introduce the concepts and foundations of our approach and the PetriCode tool as well PA-CPNs [14], the CPN sub-class that has been defined for this approach. The example is a well established and used in the literature to describe CPNs [5]. It is also a natural extension of the example we have been using in previous works [14].

This example is a simple framing protocol which is tolerant to packet loss, reordering and allows a limited number of retransmissions. The top level of the CPN model is shown in Fig. 1. The model consists of three sub-modules. Sender and Receiver represents each of the principal actors of the protocol, and Channel connects the two principals.

The protocol uses sequence numbers and a flag to indicate the last message of a frame. After a frame has been sent, the receiver, if it receives the frame, sends an acknowledgement consisting of the sequence number of the frame expected next. If the acknowledgement is not received, the sender will retransmit the frame until an acknowledgement is received or the protocol fails sending the message.

In the Sender module, shown in Fig.2, there are two sub-modules. The send sub-module is annotated with a $\langle\langle \mathsf{service} \rangle\rangle$[1] pragmatic and represents a service

[1] Pragmatics in the model and in the text are by convention written inside $\langle\langle\rangle\rangle$.

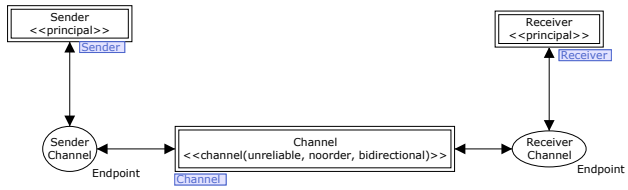


Fig. 1: The protocol system level

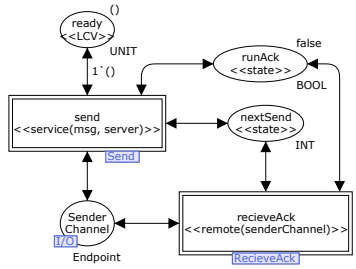


Fig. 2: The Sender principal module

provided by this principal for sending a message. The other substitution transition receiveAck, annotated with an ⟨⟨internal⟩⟩ pragmatic, represents an internal service which is to be invoked by another service of the principal. In this example, the receiveAck service is invoked from the send service.

The Sender module also contains two places, runAck and nextSend, annotated with a ⟨⟨state⟩⟩ pragmatic which contains shared data between the two services. The ready place, annotated with a ⟨⟨LCV⟩⟩ pragmatic, is used to model the life-cycle of the Sender principal and makes sure that only a single message is sent at a time.

The send service, shown in Fig. 3, starts at the transition send which opens the channel, initializes the content of the message to be sent and the sequence number. Also, at this transition, the receiveAck internal service is started by placing a token with the colour true at the ⟨⟨state⟩⟩ place runAck. The service continues from send to enter a loop at the start place. Inside the loop, the sendFrame transition retrieves the next frame to be sent based on the sequence number of the frame which is matched against the sequence number incoming from the place start. The limit place is updated with the sequence number of the current frame, and the number of times the frame has been retransmitted. Then, the current frame is sent. Due to the ⟨⟨wait⟩⟩ pragmatic at the sendFrame transition, the system waits in order to allow acknowledgements to be received. The loop ends at place frameSent. If a token is present on the place frameSent the loop will either continue with the transition nextFrame firing or end by firing the return transition. At the return transition, state places and the channel are cleared and

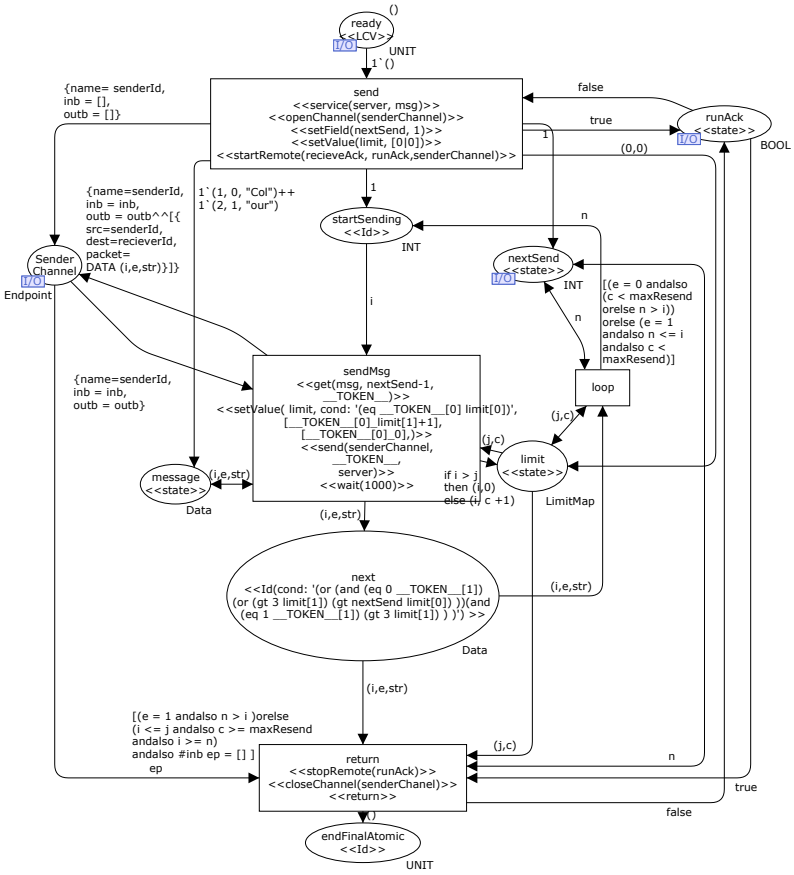


Fig. 3: The Send service module

the service terminates. In the model, we have not shown the pragmatics that can be automatically derived from the CPN model structure, see [14] for details.

The code generation approach is template-based and uses pragmatics to guide the code generation in two ways. The first way is by having structural pragmatics define the principals, services, and control-flow path within each service. The ⟨⟨principal⟩⟩, ⟨⟨service⟩⟩, and ⟨⟨Id⟩⟩ pragmatics in Figs. 1-3. The second way is to define the operations that should occur at each transition. The pragmatics are described in a domain specific language (DSL) and can often be derived from the CPN model structure. Structural pragmatics are used to generate the Abstract Template Tree (ATT), an intermediary representation of the pragmatics annotated CPN model. Each node in the ATT has pragmatics attached. Pragmatics are bound to code generation templates by template bindings. The generation

Listing 1: The Groovy template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
def ${name}(${binding.getVariables()
  .containsKey("params")
  ?params.join(", "):""}){
<%if(binding.variables
     .containsKey('pre_conds')){
  for(pre_cond in pre_conds){
    %>if(!$pre_cond) throw new
      Exception('...')
<% if(!pre_sets.contains("$pre_cond"))
   {%>$pre_cond = false<%}
  } )%>
%%yield_declarations%%
%%yield%%
<%if(binding.variables
     .containsKey('post_sets')){
  for(post_set in post_sets){
    %>$post_set = true<%
  }}%>}
```

```
def bos = new ByteArrayOutputStream()
def oos = new ObjectOutputStream(bos)
oos.writeObject(${params[1]})
_msg_ = bos.toByteArray()
DatagramPacket pack =
  new DatagramPacket(_msg_, _msg_.length,
  InetAddress.getByName(${params[2]}.host),
  ${params[2]}.port)
${params[0]}.send pack
%%VARS:_msg_%%
```

uses these bindings to generate code for each pragmatic at each ATT node. Finally, the code is stitched together using special tags in the templates.

In order to give an overview of the code generation process, we use two templates as examples. The templates are the template for the ⟨⟨service⟩⟩ pragmatic (Listing 1 (left)) and the ⟨⟨send⟩⟩ pragmatic (Listing 1 (right)).

The service template for the Groovy platform is shown in Listing 1 (left). The first line of the template creates the signature of a method what will implement the service. Lines 4 to 10 iterates over preconditions to the ⟨⟨service⟩⟩. Each precondition is checked to make sure that the service may execute. In lines 11-12 two special tags %%yield%% and %%yield_declarations%% indicates the places where the method body and the declarations will be inserted from nodes coming from the sub-nodes in the ATT.

The template for ⟨⟨send⟩⟩ is shown in Listing 1 (right). The template first creates a byte array from the data to be sent and then creates an appropriate data packet and, finally, sends the datagram packet. The template uses UDP as the underlying transport protocol, which is why the packet is created in the form of a `DatagramPacket`.

The Groovy code shown here provides a baseline implementation for the protocol. In the next section we show ho we can generate code from the same model for three other platforms.

3 Evaluating Platform Independence

In order to demonstrate the platform independence of our approach, we have generated code for the Java, Clojure and Python platforms in addition to the Groovy platform. The platforms have been chosen in order to cover three main programming languages and paradigms. Java is an imperative and object oriented programming language. Clojure is a Lisp dialect for the Java Virtual Machine (JVM). It is a functional language, however it is able to utilize Java objects

and the Java API. Python is a multi-paradigm language and, as the only language in this survey, does not rely on the JVM. Python also uses significant white-spaces which makes Python unique in this evaluation in both respects. For each of the platforms, we show selected templates corresponding to the ones shown for the Groovy platform in Sect. 2. In addition, we show an exerpt of the generated code for the Java platform since this was used as part of the evaluation of readability presented in Sect. 5

Listing 2: The Java template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
public Object ${name}(<%
def paramsVal = ""
def params2 = []
if(binding.getVariables()
   .containsKey("params")){
  params.each{
    if(it.trim() != "")
      params2 << "Object $it"
  }
  paramsVal = params2.join(", ")
}%>$paramsVal) throws Exception {
<%if(binding.variables
    .containsKey('pre_conds')){
 for(pre_cond in pre_conds){
 %>if(!$pre_cond)
  throw new RuntimeException("...");
  <%if(!pre_sets
      .contains("$pre_cond"))
    {%>$pre_cond=false;<%}
 }}%>
  %%yield_declarations%%
  %%yield%% }
```

```
1  ByteArrayOutputStream bos = new
2      ByteArrayOutputStream();
3  ObjectOutputStream oos = new
4        ObjectOutputStream(bos);
5  oos.writeObject(${params[1]});
6  byte[] _msg_ = bos.toByteArray();
7  DatagramPacket pack = new
8    DatagramPacket(_msg_, _msg_.length,
9    InetAddress.getByName((String)
10   ((Map)${params[2]}).get("host")),
11   (Integer) ((Map)${params[2]})
12   .get("port"));
13 ((DatagramSocket)${params[0]})
14 .send(pack);
```

The Java Platform. The ⟨⟨service⟩⟩ template for the Java platform is shown in Listing 2 (left). The main difference from the Groovy service template is that, in the first line, the return type and visibility protection is explicit.

The ⟨⟨send⟩⟩ template (see Listing 2 (right)) is similar to the Groovy ⟨⟨send⟩⟩ template. The differences are mainly caused by the fact that Java is explicitly typed and, at times, requires explicit casts.

Excerpts of the Java code for the Sender principal is shown in Listing 3. The first part is generated from the service template. Lines 1-5 are generated by the ⟨⟨service⟩⟩ template (Listing 2 (left)) and lines 10-17 are generated by the ⟨⟨send⟩⟩ template (Listing 2 (right)).

The Clojure Platform. The Clojure ⟨⟨service⟩⟩ template is shown in Listing 4 (left). It begins by defining a function with the name set to the name parameter. Then it creates a vector which holds incoming variables. Finally, it yields for declarations and the body of the function.

The networking templates for Clojure uses the Java networking API and the ⟨⟨send⟩⟩ template (see Listing 4 (right)) and is therefore reminiscent of Groovy

Listing 3: The Java code for the send service.

```
1  public Object send(Object msg, Object  server) throws
2        Exception { /*[msg,  server]*/ /*[Object msg, Object  server]*/
3    if(!ready) throw new RuntimeException(
4               "unfulfilled precondition: ready");
5    ready = false;
6    ...
7    __LOOP_VAR__ = true;
8    do{
9      ...
10     ByteArrayOutputStream bos = new ByteArrayOutputStream();
11     ObjectOutputStream oos = new ObjectOutputStream(bos);
12     oos.writeObject(__TOKEN__);
13     byte[] _msg_ = bos.toByteArray();
14     DatagramPacket pack = new DatagramPacket(_msg_, _msg_.length,
15         InetAddress.getByName((String)((Map)
16         server).get("host")),(Integer) ((Map)server).get("port"));
17     ((DatagramSocket)senderChannel).send(pack);
18     ...
19   }while(__LOOP_VAR__);
20   ...
21 }
```

Listing 4: The Clojure template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
(defn ${name} <%
def paramsVal = ""
def params2 = []
if(binding.getVariables().
 containsKey("params")){
 params.each{
   if(it.trim() != "") params2 << "$it"
 }
 paramsVal = params2.join(", ")
%>[$paramsVal]<%}%>
(%%yield_declarations%%
%%yield%%))
```

```
(def bos (ByteArrayOutputStream.))
(.writeObject
 (ObjectOutputStream. bos)
 @${params[1]})
(def _msg_ (.toByteArray bos))
(.send  ${params[0]}
  (DatagramPacket.
    _msg_ (alength _msg_)
    (InetAddress/getByName
    (.get ${params[2]} "host") )
    (.get ${params[2]} "port"))
)
```

and Java templates. First, the message is converted into a byte array using `java.io` streams. Then a data packet is constructed and sent using the socket given as a parameter.

The Python Platform. The Python ⟨⟨service⟩⟩ template is shown in Listing 5 (left). The template defines the method in line 2 and adds parameters, given by the template variable paramsVal in line 10. Finally, the template yields for declarations and the method body in lines 12-13.

The Python template for ⟨⟨send⟩⟩ is shown in Listing 5(right). The data using Python is a simple call to the sendto function of a socket given as params[0] with the serialized data given in params[1] and the host and port from params[2] in a tuple.

Discussion. The examples above demonstrate that our approach allows us to generate code for several platforms by providing a selection of templates for

each platform. The platforms considered, spanning several popular paradigms, gives us confidence that our approach and tool can also be applied to generate code for many other platforms. Furthermore, we are able to generate the code for each of the platforms using the same model with the same annotations and the same code generator while only varying the code generation templates and the mappings between the pragmatics and mappings between pragmatics and code templates.

Adapting the Groovy templates to Java was, for the most part simple since the two languages are similar in several respects. However, whereas Groovy is optionally typed, Java is statically typed and requires all variables to be typed or to be cast to specific types when accessing methods. Fulfilling Java's requirements for explicit types requires functionality from PetriCode so that the templates are aware of the type of variables.

Clojure is a functional language with a different control flow from languages such as Java. The main issue, compared with Groovy and Java, was related to using immutable data-structures. In Clojure all data types are, in principle, immutable. However, there is an Atom type in which values may be swapped. This was challenging because Atom values must be treated differently from pure values and lead to somewhat more verbose code than what could otherwise have been written. Also, Clojure allows the use of Java data structures, which are mutable and thus easier to work with in this case.

Python, as Groovy, is a multi-paradigm language combining the features of object oriented and functional paradigms. Creating the templates of the Python code was, although being the only language in this survey not based on the JVM, no more difficult than for the other languages. The main challenge was to handle the significant white-spaces of the Python syntax. To support this, PetriCode contains functionality to keep track of the current indentation level. This required no special treatment and was not strictly necessary, but allowed for much cleaner templates.

Table 1 shows the sizes of the Sender and Receiver principal code (measured in code lines) for each of the platforms considered. As can be seen, the code for

Listing 5: The Python template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
<%import static org.k1s.petriCode.
        generation.CodeGenerator.indent
%>${indent(indentLevel)}def ${name}(self,<%
def paramsVal = ""
def params2 = []
if(binding.getVariables()
   .containsKey("params")){
  params.each{
    if(it.trim() != "") params2 << "$it"
  }
  paramsVal = params2.join(", ")
%>$paramsVal<%}%>):
%%yield_declarations%%
%%yield%%
```

```
<%import static org.k1s.petriCode.
   generation.CodeGenerator.indent
%>${indent(indentLevel)}
   ${params[0]}.sendto(
     pickle.dumps(${params[1]}),
     (${params[2]}["host"],
     ${params[2]}["port"]))
```

Python is much smaller than the others. This is due to the efficient libraries in Python and that the Python code, for technical reasons, have much fewer blank lines which is also reflected in the templates. Table 2 shows the sizes, in lines, for selected templates and all the templates for each platform. The sizes reported are the sizes in the actual code and may not correspond to the templates as they are formatted in this paper. In this example, there was the same number of templates for each platform, but this is not necessarily always the case. As can be seen in Table 2, there is not a perfect correlation between the size of templates and the size of the generated code. This is due to, in part, some templates being more complex for some languages than others and template reuse being possible for some languages. An example is the Clojure templates, where the templates for the ⟨⟨setField⟩⟩ and ⟨⟨setValue⟩⟩ pragmatics are the same, but since the ⟨⟨setValue⟩⟩ template has more functionality than the ⟨⟨setField⟩⟩ template for all platforms, this results in a higher total number of template lines for Clojure. For each of the languages eleven new templates were constructed while ten templates were already provided as part of the PetriCode tool. The new templates were templates that are specific to the pragmatics applied for the protocol considered.

4 Evaluating Intergrateability

It should be possible to integrate code generated by our approach with existing software. We evaluate two types of integration with other software. The first type can be exemplified by having our generated cod use another library for sending and recieving data from the network. We call this type of integration downwards integration (i.e, generated code can use different third-party libraries). The other type can be exemplified by creating a runner program that employs the generated protocol for sending a message to a server. This type is called upwards integration (i.e, applications can use services provided by the generated code). We have evaluated integratability using the code generated for the Java platform based on the example in Sect. 2. However, the results are applicable for other platforms as well.

Downwards Integration. We have already shown that by changing templates, our approach can be used to generated code for different platforms. The same technique can be used to employ various libraries on the same platform to perform the same task. We illustrate this by changing the network library from the standard `java.net` library to Netty [16]. This example was chosen because

Language	Groovy	Java	Clojure	Python
Sender	131	132	119	66
Receiver	81	78	68	38
Total	212	210	187	104

Table 1: Sizes of the generated code.

Language	Groovy	Java	Clojure	Python
service	19	28	15	15
runInternal	4	10	4	3
send	9	9	8	2
All templates	154	219	251	112

Table 2: Size of code generation templates.

Listing 6: The Java template for ⟨⟨send⟩⟩ with Netty (left) and the runner for the generated Java code (right).

```
1  ByteArrayOutputStream bos =
2    new ByteArrayOutputStream();
3  ObjectOutputStream oos =
4    new ObjectOutputStream(bos);
5  oos.writeObject(${params[1]});
6  byte[] _msg_ = bos.toByteArray();
7  ((io.netty.channel.Channel)
8    ${params[0]}[0]).writeAndFlush(
9      new io.netty.channel.socket
10       .DatagramPacket(
11       io.netty.buffer.Unpooled
12       .copiedBuffer(_msg_),
13     new InetSocketAddress(InetAddress
14       .getByName((String)((Map)
15         ${params[2]}).get("host"))
16     ,(Integer)((Map)${params[2]})
17     .get("port")))).sync();
```

```
1  def sender = new Sender.Sender()
2  def reciever =
3      new Receiver.Receiver()
4  t = new Thread().start {
5    def ret = reciever.receive(31339)
6    println "Recieved: ${ret}"
7  }
8  def msg = [
9   [1,0,'Col'],[2,0,'our'],[3,0,'ed '],
10  [4,0,'Pet'],[5,0,'ri '],[6,0,'Net'],
11  [7,1,'s']
12 ]
13 sender.send(
14   msg,[host:"localhost", port:31339])
```

networking is an important function of the network protocol domain that we consider, and because Netty is substantially different from `java.net` as it is an event driven library.

Three out of twenty-one templates had to be altered to accommodate Netty as the network library for the sender principal. These were the templates that generate code for sending and receiving data from the network. We show the Netty variant of the send template from Listing 2 (right) in Listing 6 (left). The main differences is the call to the socket (or channel in the terminology of Netty) to send the message (lines 6-12).

Upwards Integration. The ability to call the generated code is necessary for the code to be useful in many instances. Our approach allows this by explicitly modelling the API in the CPN protocol model in the form of services which defines the class and method names. To demonstrate upwards integration, we have created runners for the generated implementations for each of the platforms considered. The runner for the Java platform can be seen in Listing 6 (right). This demonstrates that it is possible to use the generated services from third party software. It is worth noticing that the explicit modelling of services in the CPN model implies that it is simple to invoke the generated code.

5 Evaluating Readability

We have evaluated the readability of code generated by PetriCode in two ways. One way is that we applied a code readability metric [2] to selected snippets of the generated classes from the example described in Sect. 2, and the example described in previous works [14]. Furthermore, we have conducted a field study

The Buse-Weimer experiment (BWE)	The experiment conducted by Buse and Weimer to create the BWM. The snippets were selected from open source experiments.
The metric experiment (ME)	Our experiment to validate the results from BWE for professional developers. This experiment evaluated the twenty first snippets evaluated in the *BWE*.
The code generation experiment (CGE)	Our experiment to evaluate readability of generated code compared to non-generated code. Eight snippets were randomly selected from generated code and twelve from the open source projects in the network protocol domain.

Table 3: Overview of the experiments conducted and discussed in this section

where software engineers were asked to evaluate the readability of the generated code. This study was also used to evaluate the code readability metric.

We use the *Buse-Weimer metric* [2] (BWM) as a code readability metric. This metric was constructed by Buse and Weimer based on an experiment (the *Buse-Weimer experiment* (BWE), see Table 3) asking students at the University of Virginia to evaluate short code snippets with regards to readability on a scale of one to five. The experiment was used to construct the metric using machine learning methods to compute weights on various factors that have an impact on code readability. The final metric scores code snippets on a scale from zero to one where values close to zero indicates low readability and values close to one indicates a high degree of readability.

Our field study with software engineers took place at the JavaZone software developer conference in Oslo, Norway in September 2013. The experiment was organized into two parts. One part (*the metric experiment* (ME), see Table 3) evaluated the BWM. The other part (the *code generation experiment* (CGE), see Table 3) evaluated the readability of the generated code compared to non-generated code. Both experiments were conducted by asking software developers to evaluate twenty small code snippets with regards to readability by assigning values, on a scale from one to five, to each code snippet. The experimental set-up was created to mimic the BWE. The main advantage of our experiment over the BWE is that the dominating majority of the participants were professional software developers instead of students. The ME had 33 participants while the CGE had 30 participants.

For the CGE, we randomly selected code snippets from code generated for the Java platform based on the example described in Section 2, and the example described in [14]. We use code for the Java platform because it was used in the BWE, and the subjects of our experiments knew Java. Also, there exist several Open Source projects from which to obtain snippets for our experiments. In addition to the generated snippets, we selected, as controls, snippets from three Open Source projects in the network protocol domain. These were the Apache FtpServer, HttpCore and Commons Net [15]. All three are part of the Apache project, and we consider them to be high quality projects within the network protocol domain.

In the ME, we used the first twenty snippets from the BWE. Since we did our experiment at a conference, we could not redo the experiment with all the

Snippet	1	2	3	4	5	6	7	8	Mean	Median
Score	0,14	0,03	0,19	0,28	1,00	1,00	1,00	0,99	0,58	0,63

Table 4: The results for the BWM on generated code

Snippet	1	2	3	4	5	6	7	8	9	10	11	12	Mean	Median
Score	0,54	0,95	0,15	0,79	0,01	0,40	0,26	0,04	0,00	0,01	0,96	0,65	0,40	0,33

Table 5: The results for the BWM on selected hand-written protocol software snippets

one hundred snippets from the BWE and still expect enough software engineers to participate.

Applying the Buse-Weimer Metric. The BWM is based on the scores of hundred small code snippets. Even though the size of the snippets are not scored directly, some of the factors are highly correlated with the snippet size [11]. This makes it inappropriate to measure entire applications. Therefore, we applied the metric to the snippets selected for the CGE.

Table 4 shows the results of running the BWM tool on each of the generated code snippets. The mean and median score is above 0.5, indicating that the code is fairly readable. Also the mean and median of the generated code is higher than for the non-generated protocol-code as can be seen in Table 5.

Although the scores of the BWM on the generated code are highly encouraging, the scores are either very high or very low. This motivates an independent evaluation of the readability of the generated code (see below), as we have done with the CGE, and to validate the BWM, as we have done with the ME (see below).

The Metric Experiment: Validating the Buse-Weimer Metric. The ME was conducted to validate the BWM and the BWE. This experiment measured twenty of the code snippets that were measured in the BWE in a similar manner. The goal was to determine whether the results of the BWE holds for professional software developers.

Figure 4 shows the means of the BWE (blue/solid) and our repeat (red/-dashed) for the selected snippets. The figure suggests significant covariance even if the students in the BWE tended to judge snippets higher than the software developers in our ME. We computed three statistical tests on the correlation between the means of the two experiments (see Table 6). The correlation tests show that there is strong to medium correlations between the means and that the correlation is statistically significant($p<0.05$).The correlation tests were carried

Method	**Corrolation**	**P-value**
Pearson	cor = 0,82	$9{,}28 \cdot 10^{-06}$
Spearman	rho = 0,79	$2{,}94 \cdot 10^{-05}$
Kendall	tau = 0,61	$1{,}65 \cdot 10^{-04}$

Table 6: Correlations between the means of the ME and the BWE

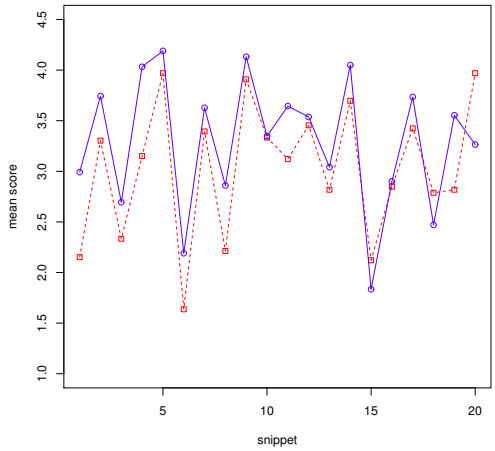


Fig. 4: The values of the selected snippets for the BWE and the ME.

out using the R [12] tool and the standard correlation test call, `corr.test()`, with all the methods available for the call. These results indicate that the BWE is relevant to professional software developers.

Mean values for the original BWE and for the ME are shown in Table 7. As can be seen, *the ME* resulted in somewhat lower scores than that of the BWE, in fact it is lower in 17 out of 20 instances. In order to determine the significance of this observation we conducted a T-test. The results of the T-test does not allow us to rule out that the means are not equal (p=0,21), although it does not give us statistically significant results on the repeat always being higher either (p=0,10), although that may be more likely.

The ME showed that there is a significant correlation between the results of the BWE (conducted with students) and the *ME* (conducted with software development professionals). This can be interpreted as evidence that the results from BWE also has validity for professional developers, although the metric based on it might be in need of some minor adjustments.

Snippet	1	2	3	4	5	6	7	8	9	10
Metric Experiment	2,15	3,30	2,33	3,15	3,97	1,64	3,39	2,21	3,91	3,33
Buse-Weimer Experiment	3,02	3,78	2,72	4,07	4,23	2,21	3,66	2,88	4,17	3,38

11	12	13	14	15	16	17	18	19	20
3,12	3,45	2,82	3,70	2,12	2,85	3,42	2,79	2,82	3,97
3,68	3,57	3,07	4,08	1,85	2,93	3,77	2,49	3,58	3,29

Table 7: Snippet means for the metric and BWE.

The Code Generation Experiment: Comparing Generated and Handwritten Code. We expected that the generated code would not do quite as well as the hand-written high-quality code used as control. Therefore, our hypothesis was that the generated code would be within the standard deviation of the hand-written written code. The mean score for each of the snippets in the CGE are shown in Table 8. Snippets one to eight are generated code while snippets eight to twenty are hand-written. To check our hypothesis, we ran Welch's T-test on the results which is useful for determining the difference between the two samples. The first hypothesis we checked was whether the generated snippets are less readable than the hand-written ones. The results of a Welch's Two Sample t-test showed that the generated code-snippets scored below that of the hand-written code ($p=4{,}81 \cdot 10^{-05}$).

Then we checked the hypothesis when reducing the score of the measurements from the Apache projects by the standard deviation of those measurements. The Welch's Two Sample t-test indicates that the generated code scores better than one standard deviation below the hand-written code (p=0,03). This indicates that our goal of being readable within a standard deviation of non-generated code is met both by measuring via the BWM and experimentally.

Table 9 shows normalized means for each snippet from the CGE and the results of running the BWM on the corresponding snippets. As can be seen, the correlation is less than strong as confirmed by running correlation tests (see Table 10). Even though the results of the ME indicates that the BWE, which the BWM is based on, is valid even for professional software developers, we content that the results of the CGE are more reliable than the BWM. This is because the BWM is derived from software from different domains and that it is sensitive to snippet length. This indicates that the BWM is not relevant to code for network protocols.

Assessment of Validity of Our Results. As with most experimental approaches, this evaluation has some threats to the validity of the results. These are issues we have identified that might skew our results. One such threat to validity for the original BWE was that they used student as subjects who may or may not disagree with professional software developers on the readability of code. We have tried to alleviate this threat in the ME by repeating part of the BWE with professional developers. Further threats to validity to the experiments and results described in this section are discussed in the following.

Small sample size and limited number of participants may skew the results. Since we conducted this experiment at a software developer conference where people tended to be on their way to some lecture, we had to limit the number

Snippet	1	2	3	4	5	6	7	8	9	10
Mean	2,40	2,10	3,83	3,23	2,67	3,13	2,97	2,73	3,90	3,97

11	12	13	14	15	16	17	18	19	20
2,67	3,13	2,73	3,43	3,67	3,07	3,83	2,00	3,20	3,93

Table 8: Means of results for generated code (1-8) and Apache projects code (9-20).

Snippet	1	2	3	4	5	6	7	8	9	10
Experiment Score	0,48	0,42	0,77	0,65	0,53	0,63	0,59	0,55	0,78	0,79
Metric Score	0,14	0,03	0,19	0,28	1,00	1,00	1,00	0,99	0,54	0,95
	11	12	13	14	15	16	171	18	19	20
	0,53	0,63	0,55	0,69	0,73	0,61	0,77	0,40	0,64	0,79
	0,15	0,79	0,01	0,40	0,26	0,04	0,00	0,01	0,96	0,65

Table 9: Normalized means from the CGE and results from applying the BWM

Method	**Corrolation**	**P-value**
Pearson	cor = 0,21	0,37
Spearman	rho = 0,20	0,40
Kendall	tau = 0,14	0,40

Table 10: Correlation between normalized experimental scores and the BWM applied to the same snippets

of snippets we asked each participant to evaluate. Also, because professional software developers are harder to recruit than students, the number of participants was limited. Furthermore, it is possible, albeit unlikely, that the people participating in the experiment are not representative for software developers as a whole. These threats can be alleviated by conducting broader studies on larger groups of developers and using interviews.

In our experiments, we used small randomly selected code snippets as proxies for code readability. We do this both for practical and conceptual reasons. The practical reasons revolve around what we realistically could expect participants to score. If they had to read entire classes or software projects in order to score the code, this would have taken to much time and could have resulted in getting too few participants in our experiments. Furthermore, we wanted to evaluate the BWM since it is the only implemented metric we could find in the literature. The more conceptual reason is that if each snippet is readable, then the whole code is likely to be readable as well. In our approach, high-level understanding is based more on the CPN models of the protocols than on the implementation, so it makes sense for us to concentrate on low level, snippet-sized readability, since readability in the large is intended to be considered at the level of the model.

6 Conclusions and Related Work

In this paper, we have evaluated our code generation approach and supporting software, with respect to platform independence, the integratability of the generated code as well as the readability of the generated code.

Platform independence was evaluated by generating code for a protocol for three platforms in addition to the Groovy platform from a single CPN model. The number of and differences between the platforms gives us confidence that our approach and the PetriCode tool can be used to generate protocol implementations for many target platforms. All the platforms considered have automatic memory management in the form of garbage collection. This is convenient, but

we intend to support platforms without automatic memory management in the future.

Platform independence is especially important for network protocols since they are used to communicate between two or more hosts that often run on different underlying platforms. Although there exists many tools that allow generating code from models claiming to be platform independent, few studies seem to have been made actually generating code for several platforms.

MDA [8] and associated tools rely on different platform specific models (PSM) to be derived for platforms before generating code for each platform. This adds an extra modelling step compared to our approach and may require somewhat different PSMs for different platforms. The Eclipse Model To Text (M2T) [3] project provides several template languages for code generation from Ecore models. In general, M2T languages can generate code for several platforms. However, to go beyond pure structural features and standard behaviour, the developer must create customized code generators. In [9] code is for protocol is generated using UML stereotypes and various UML diagram types. The UML diagrams, annotated with stereotypes according to a custom made UML profile, combined with a textual language named GAEL are used to obtain protocol specification in the Specification and Description Language (SDL) [1, 4]. The authors also conjecture that the approach can be used to generate code for any platform. The use of stereotypes in the approach presented in [9] is similar to the pragmatics that our approach uses. However, a difference is that several diagram types are used in the UML based approach in contrast to our approach where we use CPNs to describe both structure and behaviour.

MetaEdit+ [17] allows code generation of visual Domain Specific Modelling Languages (DSMLs). MetaEdit+ and the DSML approach is similar to the PetriCode approach since CPNs and pragmatics constitute a DSML. A main difference is that MetaEdit+ allows users to generate custom graphical languages while PetriCode uses CPN, but extends CPNs using pragmatics. This allows us to use the properties of CPNs for verification and validation, and also to use a single syntax for different domains.

The Renew [7] tool uses a simulation-based approach where annotated Petri Nets can be run as stand-alone applications. The simulation-based approach is fundamentally different from our approach where the generated code can be inspected and compiled in the same way as computer programs created with traditional programming languages. A detailed comparison between these two approaches would be an interesting avenue for future work.

We evaluated the integratability of the generated code in two directions: upwards and downwards integratability. Upwards integratability was evaluated by showing that the generated protocol software can be called by programs running the protocols. Downwards integratability was evaluated by showing how we can change the network API for the Java platform by binding different templates to some of the pragmatics.

Readability of the generated code was evaluated by an automatic metric and an experiment. According to the BWM, the generated code is as, or possibly

even more, readable than the samples of high quality code in the same domain that we used for comparison. Based on our experiment with software developers, however, the generated code is somewhat less readable but within a standard deviation of the non-generated code. A contribution of this paper is also to provide evidence that the experimental results from the BWE are relevant to professional software developers in addition to the students. However, based on the discrepancy between the experimental evaluation, it seems that the BWM may not be applicable to code in the network protocol domain. To the best of our knowledge, there are no previous work evaluating intergrateability and readability of automatically generated software.

In the future we will evaluate the verifiability of the models used in our approach by applying verification techniques to example protocols. We also intend to develop a set of template libraries that can be used for code generation as well as procedures for testing code generation templates. Another possible direction for future work is to apply our code generation approach to other domain.

References

1. F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *Communications Surveys Tutorials, IEEE*, 4(1):2–20, 2002.
2. R.P.L. Buse and W.R. Weimer. A metric for software readability. In *Proc. of ISSTA'08*, pages 121–130, NY, USA, 2008. ACM.
3. IBM. *Eclipse Model To Text (M2T)*. `http://www.eclipse.org/modeling/m2t/`.
4. ITU-T. Recommendation z.100 (11/99) specification and description language (sdl), 1999.
5. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
6. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *ToPNoc VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
7. O. Kummer et al. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In *Proc. of ICATPN '04*, volume 3099 of *LNCS*, pages 484–493. Springer, 2004.
8. Object Management Group. *MDA Guide*, June 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.
9. J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. UML for protocol engineering-extensions and experiences. In *Proc. of TOOLS '00*, pages 82–93, 2000.
10. PetriCode. *Example protocol*. `http://bit.ly/19HU8U4`.
11. D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proc. of MSR '11*, pages 73–82. ACM, 2011.
12. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2013.
13. K. I. F. Simonsen. Petricode: A tool for template-based code generation from cpn models. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods*, volume 8368 of *LNCS*, pages 151–163. Springer, 2014.
14. K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.

15. The Apache Software Foundation. FtpServer http://mina.apache.org/ftpserver-project/, HttpCore https://hc.apache.org/, Commons Net http://commons.apache.org/proper/commons-net/.
16. The Netty project. *Netty.* http://netty.io.
17. J.P. Tolvanen. MetaEdit+: domain-specific modeling for full code generation demonstrated. In *Proc of SIGPLAN '04*, pages 39–40. ACM, 2004.

Chapter 13

Implementing the Web Socket Protocol based on Formal Modelling and Automated Code Generation

Implementing the WebSocket Protocol based on Formal Modelling and Automated Code Generation

Kent Inge Fagerland Simonsen[1,2] and Lars Michael Kristensen[1]

[1] Department of Computing, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
[2] DTU Compute, Technical University of Denmark, Denmark
Email: {kisi}@dtu.dk

Abstract. Model-based software engineering offers several attractive benefits for the implementation of protocols, including automated code generation for different platforms from design-level models. In earlier work, we have proposed a template-based approach using Coloured Petri Net formal models with pragmatic annotations for automated code generation of protocol software. The contribution of this paper is an application of the approach as implemented in the PetriCode tool to obtain protocol software implementing the IETF WebSocket protocol. This demonstrates the scalability of our approach to real protocols. Furthermore, we perform formal verification of the CPN model prior to code generation, and test the implementation for interoperability against the Autobahn WebSocket test-suite resulting in 97% and 99% success rate for the client and server implementation, respectively. The tests show that the cause of test failures were mostly due to local and trivial errors in newly written code-generation templates, and not related to the overall logical operation of the protocol as specified by the CPN model.

1 Introduction

The vast majority of software systems today can be characterised as concurrent and distributed systems as their operation inherently relies on protocols executed between independently scheduled software components and applications. The engineering of correct protocols can be a challenging task due to their complex behaviour which may result in subtle errors if not carefully designed. Furthermore, ensuring interoperability between independently made implementations is also challenging due to ambiguous protocol specifications. The use of formal modelling in combination with verification and model checking provides a prominent approach to the development of reliable protocol implementations.

Coloured Petri Nets (CPNs) [8] is formal language combining Petri Nets with a programming language to obtain a modelling language that scales to large systems. In CPNs, Petri Nets provide the primitives for modelling concurrency and synchronisation while the Standard ML programming language provides the primitives for modelling data and data manipulation. CPNs have

been successfully applied for the modelling and verification of many protocols, including Internet protocols such as the TCP, DCCP, and DYMO protocols [2, 11]. Formal modelling and verification have been useful in gaining insight into the operation of the protocols considered and have resulted in improved protocol specifications. However, earlier work has not fully leveraged the investment in modelling by also taking the step to automated code generation as a way to obtain an implementation of the protocol under consideration.

In earlier work [15], we have proposed the PetriCode approach and developed a supporting software tool [17] for automatically generating protocol implementations based on CPN models. The basic idea of the approach is to enforce particular modelling patterns and annotate the CPN models with code generation *pragmatics*. The pragmatics are bound to code generation templates and used to direct the model-to-text transformation that generates the protocol implementation. As part of earlier work, we have demonstrated the use of the PetriCode approach on small protocols. In addition, we have shown that our approach supports code generation for multiple platforms, and that it leads to code that is readable and also upwards and downwards compatible with other software [16].

The main contribution of this paper is to demonstrate that our approach and tool scale to support an industrial-sized protocol by automatically generating code for the WebSocket [5] protocol for the Groovy [7] platform. The WebSocket protocol is a relatively new protocol currently under development by the IETF. The WebSocket protocol makes it possible to upgrade an HTTP connection to an efficient message-based full-duplex connection. The WebSocket protocol address the performance problems of the HTTP protocol caused by the request-response interaction model and verbose headers. This is done by allowing HTTP to upgrade to a WebSocket connection in which a session is kept alive and messages may be transmitted in both directions freely with much lower overhead than with HTTP. WebSocket has already become a popular protocol for several web-based applications where bi-directional communication with low latency is needed such as games and media streaming services. The contributions of this paper include showing how we have been able to model the WebSocket protocol following the PetriCode modelling conventions, and to verify the model through state space exploration. Furthermore, we demonstrate in this paper that the generated code is interoperable with other WebSocket implementations, and we test the our implementation using the Autobahn WebSocket test-suite [18].

Outline. Section 2 presents the CPN model of the WebSocket protocol. In Sect. 3 we show how state space exploration was used to verify the operation of the model focusing on the proper establishment and termination of WebSocket connections. Section 4 describes the procedure to generate an implementation of the WebSocket protocol from the CPN model using the PetriCode tool. In Sect. 5, we present the results from testing the generated code by showing that it is interoperable with other WebSocket implementations and by employing the Autobahn WebSocket test-suite. Finally, in Sect. 6 we provide a discussion of related work, and sum up the conclusions and directions for future work. Due to space limitations, we refer to [8] for a detailed introduction to CPN concepts.

2 The CPN WebSocket Code Generation Model

The CPN model of the WebSocket protocol follows the structure imposed by our code generation approach, and consists of a set of modules hierarchically organised into three levels: the protocol systems level, the principal level, and the service level. In the following, we present representative parts of the CPN model which was constructed using CPN Tools [4].

Figure 1 shows the top-level module of the CPN model constituting the protocol system level. The protocol system consists of a Client and a Server principal as modelled by the two accordingly named *substitution transitions* drawn as rectangles with a double-lined border. These two substitution transitions are annotated with the ⟨⟨principal⟩⟩ code generation pragmatic to denote that they represent protocol principals. The Channel substitution transition annotated with the ⟨⟨channel⟩⟩ pragmatic represents the channel connecting the two principals. The two substitution transitions are connected by *places* (drawn as ellipses) modelling send and receive buffers for the client and server. The rectangular tags attached to the substitution transitions specify the name of the *submodule* which refines the compound behaviour represented by the substitution transition.

The Client principal level module is depicted in Fig. 2. It is the submodule associated with the Client substitution transition in Fig. 1. The principal level makes explicit the *services* offered by the principal by means of the ⟨⟨service⟩⟩ and ⟨⟨internal⟩⟩ pragmatics attached to substitution transitions. The ⟨⟨service⟩⟩ pragmatic is used to denote substitution transitions where the attached submodule represents a service that is intended to be used by the application employing the protocol. Substitution transitions annotated with ⟨⟨internal⟩⟩ represent services that are used internally in the protocol principal. It can be seen that the client has six external and two internal services. A principal level module also models the *internal state* of the principal via places annotated with the ⟨⟨state⟩⟩ pragmatic, and captures the life-cycle of the principal via places annotated with the ⟨⟨LCV⟩⟩ pragmatic. The life-cycle determines the possible orders in which the services can be invoked. Initially, the only ⟨⟨LCV⟩⟩-annotated place that contains a token is the READY place (top) which enables the OpenConnection service. After the OpenConnection service has completed, there will be a token on the OPEN place, and all the external services (except OpenConnection) will be enabled allowing the employing application to send and receive messages, send

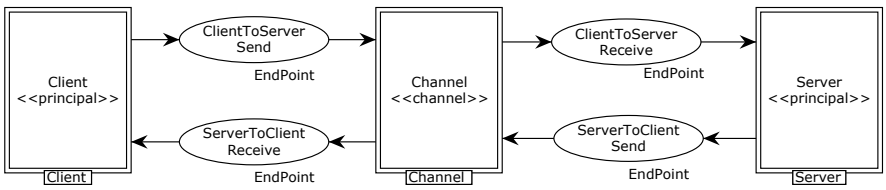


Fig. 1. The top level of the WebSocket protocol model

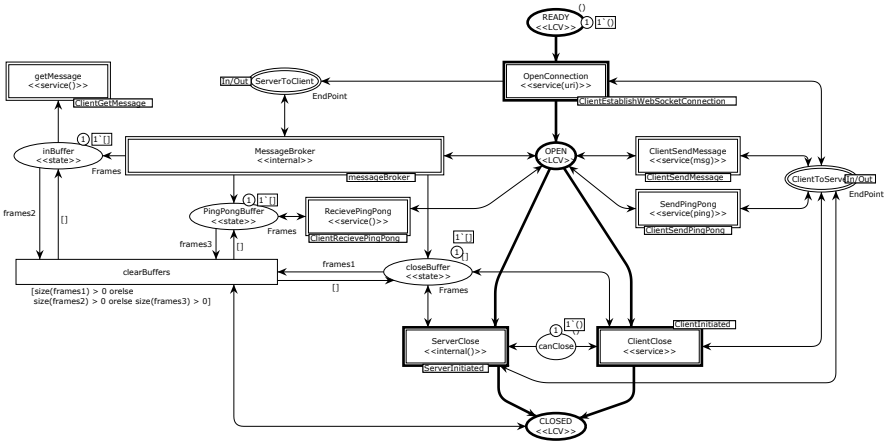


Fig. 2. The Client principal module

and receive ping and pong messages, and close the connection. The exchange of ping and pong messages provides a keep-alive mechanism in the protocol.

The MessageBroker module which is the submodule of the MessageBroker substitution transition is shown in Fig. 3. The MessageBroker is an example of an internal service. It is responsible for dispatching the incoming messages into the appropriate buffer represented in the module by places annotated with the ⟨⟨state⟩⟩ pragmatics. There is one such buffer for each of the message types: the inBuffer keeps text and binary messages, the pingpongBuffer keeps ping and pong messages, the closeBuffer keeps the closing messages while the fragments place keeps frames of messages that have not yet been completely received. The messages are dispatched by inspecting the type of the messages. The MessageBroker internal service is enabled when the WebSocket connection is in an OPEN state and the module captures the control flow in dispatching received messages as indicated by the places annotated with an ⟨⟨Id⟩⟩ pragmatic. The execution of the service starts at the transition ReceiveDataFrame. Then it enters a loop starting at place wait receive. At the transition receive a new frame is received. This is modelled as a single operation to keep the model at a high level of abstraction. This means that the details of actually receiving a message must be encoded in the code generation template associated with the ⟨⟨receive⟩⟩ pragmatic. If the frame is the last frame of a fragmented message, the entire message is reconstructed. Next, there is a branch in the model based on the Fin and OpCode fields in the frame. The message is dispatched to either the inBuffer (data), PingPongBuffer, closeBuffer, or nonFinal. After the message or frame has been dispatched, the branches merge before the next iteration.

The getMessage service shown in Fig. 4 returns the next message in the buffer. This service will be used to illustrate code generation in Sect. 4.

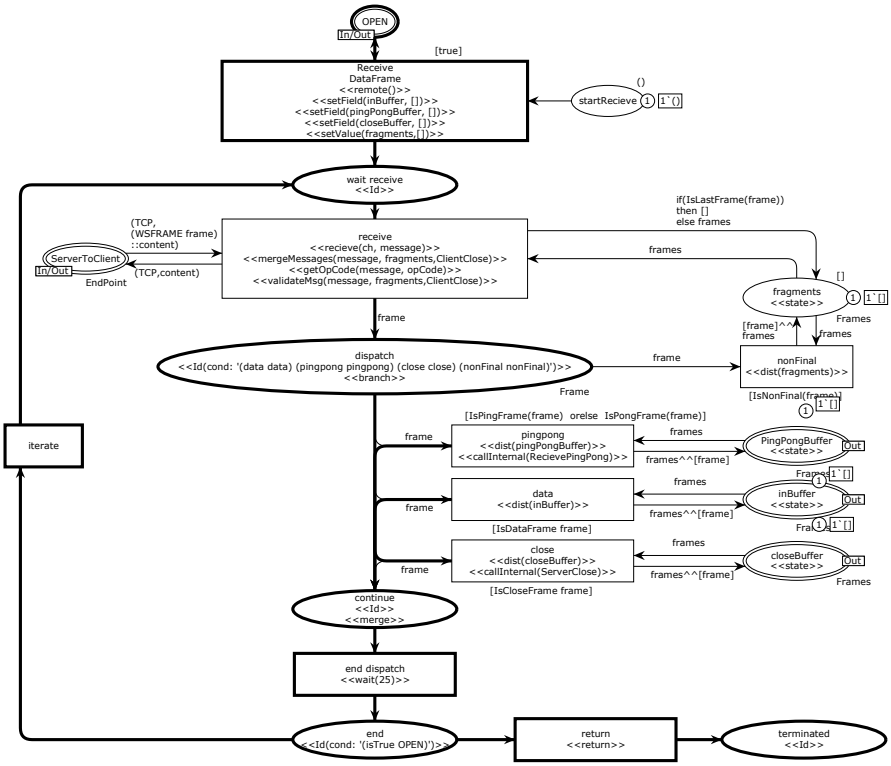


Fig. 3. The MessageBroker internal service

This is an example of a service with only a single transition. The transition is annotated with the ⟨⟨service⟩⟩, ⟨⟨getMessage⟩⟩ and ⟨⟨return⟩⟩ pragmatics. This models that the service is first entered, then the ⟨⟨getMessage⟩⟩ operation is performed, and the service terminates. The transition getMessage is only enabled when there is at least one message in the message buffer as modelled by the place inBuffer.

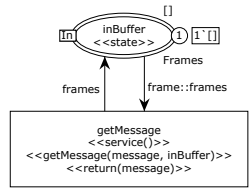


Fig. 4. The getMessage service

The complete CPN model consists of 19 modules. Each of the two principals have eight sub-modules which all correspond to the external and internal services in the protocol. In total, the model consists of 136 places and 84 transitions. This reflects the complexity of the protocol, but also the high-level nature of the model which has been important in keeping the number of elements manageable.

3 CPN WebSocket Model Verification

CPNs have a formal semantics which makes it possible to conduct model simulation and model checking (verification) prior to code generation. This is a major advantage of an approach based on a formal modelling language as this can be used to eliminate design errors prior to code generation and testing of the generated protocol implementation. CPN Tools used for construction of the WebSocket CPN model supports model checking of behavioural properties by means of *state space exploration*. The basic idea of state space exploration is to explore all the reachable states of the model to determine whether a model satisfies a given property or not. This means that state space exploration will exhaustively explore (test) all the possible executions of the CPN model. As the CPN model specifies the behaviour of both the client and the server, the state space exploration exercises the client against all the possible behaviours of the server and visa versa.

Our aim has been to apply state space exploration of the CPN model as a first test to eliminate possible errors in the logical specification of the WebSocket protocol. For this, we adopted a lightweight approach where we consider the following behavioural properties **P0**, **P1** and **P2** of the CPN model:

P0 From the initial state it is possible to reach states in which the WebSocket connection has been opened (i.e., both the client and the server are in the *open* state). In the model this means that the places names OPEN in the Client and the Server modules each have one token and none of the other places other places modelling the life-cycle of the principals have a token. It should be noted that we cannot establish that the WebSocket connection will eventually be opened since the server side may initiate a close before the client side is in the *open* state.

P1 All terminal states (i.e., states without enabled transitions) correspond to states in which the WebSocket connection has been properly closed (i.e., both the client and the server are in the *closed* state). In the model this means that the places named CLOSED in the Client and Server modules each have one token and that none of the other places modelling the life-cycle of the principals have a token.

P2 From any reachable state, it is always possible to reach a state in which the WebSocket connection has been properly closed. This means that independently of how messages are exchanged, it is always possible to properly close the WebSocket connection.

In order to check that in all terminal states both the client and the server are in the *closed* state, we wrote a simple query in the Standard ML language using functions that are built into CPN Tools. The query can be seen in Listing 1.1.

The functions `server_open` and `client_open` are predicates for the client and server that take a state as argument and return true if and only if the principal is in an open state, i.e., the place OPEN has one token, and all the other LCV places have no tokens. The functions `server_close` and `client_close`

Listing 1.1. The queries used to verify properties P0-P2.

```
fun client_open (n) = (State.Client'CLOSED 1 n) = [] andalso
 (State.Client'OPEN 1 n) = [()] andalso (State.Client'READY 1 n) = [];

fun server_open (n) =
 (State.Server'CLOSED 1 n) = [] andalso (State.Server'OPEN 1 n) = [()] andalso
 (State.Server'Idle 1 n) = []   andalso (State.Server'READY 1 n) = [];

fun client_closed (n) = (State.Client'CLOSED 1 n) = [()]
 andalso (State.Client'OPEN 1 n) = [] andalso (State.Client'READY 1 n) = [];

fun server_pred (n) =
 (State.Server'CLOSED 1 n) = [()] andalso (State.Server'OPEN 1 n) = [] andalso
 (State.Server'Idle 1 n) = []     andalso (State.Server'READY 1 n) = [];

fun IsProperOpen(n) = server_open(n) andalso client_open(n);
fun IsProperClosed(n) = server_closed(n) andalso client_closed(n);

PredAllNodes(IsProperOpen) <> []              (* property P0 *)
List.all IsProperClosed (ListTerminalStates()); (* property P1 *)
HomeSpace(ListTerminalStates());                (* property P2 *)
```

Table 1. Results of verification of the WebSocket CPN model

Client Messages	Server messages	Nodes	Arcs	Time (secs)	Terminal states
yes	no	2747	9,544	1	2
no	yes	2867	9,956	2	2
yes	yes	39189	177,238	246	4

are similar for the case of closed. The predicates are used to obtain the predicates `isProperOpen` and `isProperClosed` that characterises properly open and closed states, respectively. The property P0 is checked using the query function `PredAllNodes` which returns all states satisfying a given predicate (in this case `IsProperOpen`). It is then checked whether the resulting list of states is non-empty. For establishing P1, we check that all terminal states in the state space which are returned by the built-in query function `ListTerminalStates` satisfies the `IsProperClosed` predicate. Finally, P2 is checked using the query function `HomeSpace` which checks if the list of nodes provided constitute a *home space*, i.e, constitute a set of states where at least one of the states can always be reached.

Table 1 summarises the results from the verification. We have considered three possible configurations of the model. One where the client sends data to the server; one where the server sends data to the client; and one where both the client and the server sends data. The table lists the number of Nodes and Arcs in the state space, the amount of Time used to generate the state space, and the number of Terminal States. For all configurations, we were able to establish the properties P0, P1 and P2 which provides confidence in the correctness of the model. During the verification process, several minor modelling errors were identified and fixed. For example, this lead to the inclusion of the clearBuffers transition (see Fig. 2) which was added to properly clean up message buffers and reduce the number of terminal states.

The major drawback with state space exploration techniques is the state explosion problem which means that the state space in many cases grows too large to be handled with the available computing power. It is interesting to observe that the size of the state space for the model described in Sect. 2 is relatively small for the configurations considered. This shows how our modelling approach makes it possible to construct models at a high-level of abstraction so that it is feasible to fully verify even industrial-sized protocols.

4 Automated Code Generation

In this section we describe the code generation process for the WebSocket protocol targeting the Groovy platform and illustrate it with examples of code generation templates and code snippets. Groovy is also the implementation language of the PetriCode tool [17] and has been chosen because it has several features that makes it easy to implement and debug templates including dynamic typing, closures and iterators.

The automatic code generation process, as implemented in the PetriCode tool, starts with a CPN model annotated with pragmatics. The model is first transformed into an intermediary representation in the form of an *abstract template tree* (ATT). The ATT reflects the hierarchical structure of the CPN model down to the service level. On the service level, the ATT contains *blocks* that are derived from the control flow path specified by the ⟨⟨id⟩⟩ pragmatics of the service level modules. The next step in the code generation process is to traverse the ATT and emit code for each node by applying code generation templates bound to the pragmatics of a node. Pragmatics are bound to templates using *template bindings* which are defined in a domain specific language (DSL). When this is done, the code is stitched together using special markers in the generated code. The details are described in [15, 17]. Our approach makes it possible to produce code for several platforms and programming languages. This is achieved by using different sets of code generating templates and binding them as appropriate to code generation pragmatics through the use of the DSL.

When the code generator, on its traversal through the ATT, encounters a node annotated with a ⟨⟨principal⟩⟩ pragmatic it executes the associated templates which, in the Groovy platform, defines a class. Then the traversal continues to the child nodes of the principal. When the generation traverses child nodes of a principal and encounters a node containing a ⟨⟨service⟩⟩ or ⟨⟨internal⟩⟩ pragmatic it executes the service template. The code generation for the principal is completed by replacing a special tag, %%yield%% with the result for the service template for all underlying services. The generated code of the client with declaration and method bodies omitted is shown in Listing 1.2. As can be seen when comparing with Fig 2, there is one method defined for each external and internal service. This comprises the API for the WebSocket client implementation with all the callable methods and their signature.

The template for the ⟨⟨service⟩⟩ pragmatic is shown in Listing 1.3. Lines 1-2 define a new method and its signature. The lines 3-12 set up preconditions (if

Listing 1.2. The generated code for the services in the client.

```
1  class Client {
2  ...
3  def MessageBroker(){ ... }
4  def ServerClose(){ ... }
5  def OpenConnection(uri){ ... }
6  def ClientSendMessage(msg){ ... }
7  def ReceivePingPong(){ ... }
8  def SendPingPong(ping){ ... }
9  def ClientClose(){ ... }
10 def getMessage(){ ... }
11 }
```

Listing 1.3. The template bound to the ⟨⟨service⟩⟩ pragmatic

```
1  def ${name}(${binding.getVariables()
2              .containsKey("params") ? params.join(", ") : ""}){
3  <%
4    if(binding.variables.containsKey('pre_conds')){
5      for(pre_cond in pre_conds){
6        %>if(!$pre_cond) throw
7            new Exception('unfulfilled precondition: $pre_cond')
8        <%
9        if(!pre_sets.contains("$pre_cond")){%>$pre_cond = false<%}
10     }
11   }
12 %>
13   %%yield_declarations%%
14   %%yield%%
15   <%if(binding.variables.containsKey('post_sets')){
16     for(post_set in post_sets){
17       %>$post_set = true<%
18      }
19  }%>
20 }
```

applicable) based on the manipulation of places at the service level that are annotated with the ⟨⟨LCV⟩⟩ pragmatic. The next two lines are place-holder tags that show where declarations and the method body will be inserted respectively. Finally, post-conditions are set and the method body ends in line 20.

Listing 1.4 shows the template for the ⟨⟨getMessage⟩⟩ pragmatic used on the transition in Fig. 4. The template takes two parameters. The name of variable to set the next message to, and the name of the buffer to retrieve the next message from. First, the template checks to see if the buffer is not empty. If it is not empty, the first message is retrieved from the buffer. Then the payload is translated into a `String` or a byte array depending on the message type and the variable given in the first parameter to the pragmatic is set to the payload of the message. If the buffer is empty the variable given in the first parameter to the pragmatic is set to *null*.

The generated code for the getMessage service is shown in 1.5. Lines 1-4 and 21 are generated by the service template. The rest of the code, except from the return line, follows the template for ⟨⟨getMessage⟩⟩ where the first and second

Listing 1.4. The template for the ⟨⟨getMessage⟩⟩ pragmatic.

```
if(${params[1]} != null &&  ${params[1]}.size() > 0){
  ${params[0]} = ${params[1]}.remove(0)
  byte[] bArr = new byte[${params[0]}.payLoad.size()]
  for(int i = 0; i < bArr.length; i++){
    bArr[i] = ${params[0]}.payLoad.get(i)
  }
  if(${params[0]}.opCode == 1){
    ${params[0]} = new String(bArr)
  }else if(${params[0]}.opCode == 2) {
    ${params[0]} = bArr
  }
}else{
  ${params[0]} = null
}
%%VARS: ${params[0]}, ${params[1]}%%
```

Listing 1.5. The generated code for the getMessage service in the client.

```
def getMessage(){
  /*vars: [__TOKEN__:, message:]*/
  def __TOKEN__
  def message
  //getMessage
  if(inBuffer != null && inBuffer.size() > 0){
    message = inBuffer.remove(0)
    byte[] bArr = new byte[message.payLoad.size()]
    for(int i = 0; i < bArr.length; i++){
      bArr[i] = message.payLoad.get(i)
    }
    if(message.opCode == 1){
      message = new String(bArr)
     }else if(message.opCode == 2) {
       message = bArr
     }
  }else{
    message = null
  }
  return message
}
```

parameters have been replaced with inBuffer and message respectively since those are the two parameters given to the pragmatic in Fig. 4.

In order to generate code for the WebSocket protocol, we reused 10 templates from the library of templates provided by PetriCode. In addition, 22 new templates were needed, including two templates that override existing templates. New templates were needed because the WebSocket protocol has many features we have not encountered with earlier examples, such as receiving and interpreting binary messages, and validating handshakes and frames.

5 Testing the Generated WebSocket Implementation

We validated the operation and interoperability of the generated code in two ways. First, we created test drivers for the generated WebSocket implementation to connect to the example chat server and client [1] that comes with the GlassFish

Listing 1.6. The code for the client runner.

```
def client = new Client()
client.OpenConnection(new URI("ws://localhost:31337/chat/websocket"))
def t = Thread.start {
  while(true){
    def msg =  client.getMessage()
    if(msg) println "RECEIVED: $msg"
    Thread.sleep(1000)
}}
client.ClientSendMessage("${args[0]} joined")
BufferedReader br = new BufferedReader(new InputStreamReader(System.in))
while(true){
  print "#: "
  def msg = br.readLine()
  if(msg == "#quit"){
    client.ClientClose()
    try{ Thread.wait(1000) }
    catch(Exception ex){  }
    System.exit(0)
  } else if(msg == "#close"){
    client.ClientClose()
  }else{
    client.ClientSendMessage("${args[0]}: $msg")
  }
}
```

Application Server [14]. Secondly, we submitted the generated implementation to the Autobahn Testsuite [18] version 0.5.5[1].

Chat Application. The code for the chat client using the generated API (cf. Listing 1.2) is shown in Listing 1.6. The chat client uses the generated WebSocket protocol as an API given the signatures of the principal level. The client begins by creating a `Client` object from the generated code and opens a WebSocket connection to the server. Then, a thread is started to receive messages which polls the client object for new messages and prints any received messages to the console. After the message receiving thread is started, the client sends a message notifying the server that the client has joined the chat. Finally, the client enters an infinite loop that listens to the console for messages and sends any messages to the server. The server is implemented in a similar way using the generated `Server` class as the server-side WebSocket API.

Listing 1.6 demonstrates that our approach is upwards compatible, i.e., that the services of the generated code can easily be used by third party software. A key feature that provides this is that we include the API in the model as the services at the principal level of the CPN model.

Figure 5 shows the chat client (upper right) and server (lower right) running together with the web-based chat client from [1]. The web-based client has only been modified to connect to the server using the generated API by changing a hard-coded server address. We also tested that the chat client is able to connect and communicate with a chat-server from [1].

[1] The test results can be seen at http://t.k1s.org/wsreport

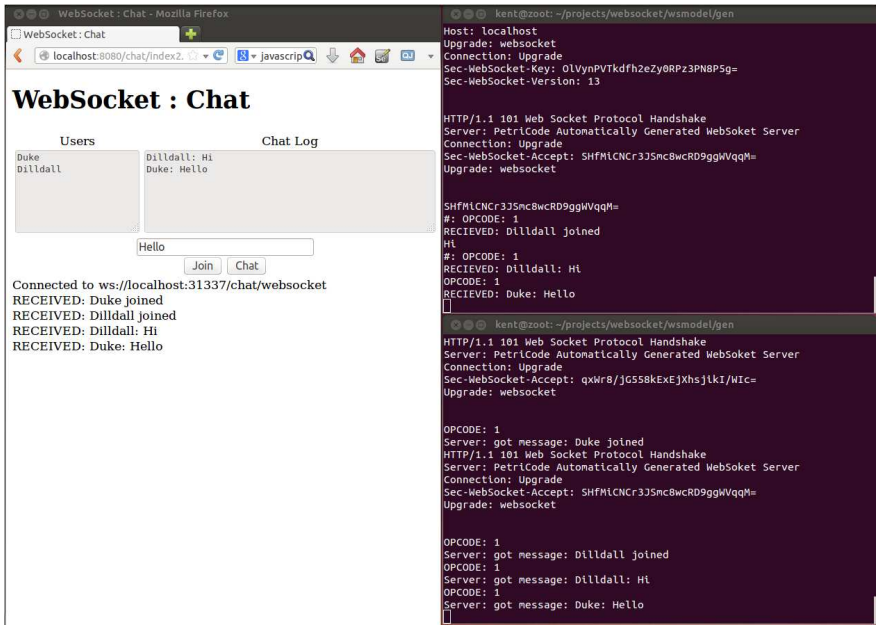


Fig. 5. Chat server and client using the generated API (right) and a web-based chat client connected to the same server (left)

Autobahn Test-suite. The Autobahn WebSocket test-suite provides comprehensive validation of server and client implementations of the WebSocket protocol. The test-suite has been used by several high-profile projects to develop and validate WebSocket implementations including the Firefox and Jetty projects. When running the Autobahn test-suite several problems with the implementation were discovered. Most of the problems were simple oversights in the code generation templates that were easily fixed once they were identified. An example of the trivial problems that were not evident when running the chat application was that the HTTP header lines were terminated with LF instead of the mandated CRLF. However, one change to the CPN model was necessary. This was related to fragmented messages where we added a buffer for temporarily storing frames of unfinished messages and a transition to distributing non-final frames. This was necessary because a WebSocket endpoint should be able to handle control messages intermingled with fragmented messages. The new elements, which can be seen in Fig. 3, are the place fragments, the transition nonFinal, and the arcs connected to those two elements.

A summary of the result for the final Autobahn tests can be seen in Table 2. The Autobahn test suite contains 301 tests cases for the client and server. For the client, 10 test cases fail and for the server, 4 test cases fail. The extra test cases that fail on the client concern performance with large messages. The test cases that fail for both the server and client are UTF-8 parser errors. This is because

the Java implementation of UTF-8 parsers is more lenient than the Autobahn test-suite expects. Therefore, we had to create our UTF-8 validator which fails to identify some UTF-8 errors.

Table 2. Results for the Autobahn tests

Tests	Server Passed	Client Passed
1. Framing (text and binary messages)	16/16	16/16
2. Pings/Pongs	11/11	11/11
3. Reserved bits	7/7	7/7
4. Opcodes	10/10	10/10
5. Fragmentation	20/20	20/20
6. UTF-8 handling	137/141	137/141
7. Close handling	38/38	38/38
9. Limits/Performance	54/54	48/54
10. Auto-Fragmentation	1/1	1/1

6 Conclusions and Related Work

In this paper we have shown that the PetriCode code generation approach can be applied to industrial sized protocols as exemplified by the WebSocket protocol. Obtaining the implementation was achieved with limited effort even though quite a few new templates were created. We have found that the template provides an effective way to force the code to be modular. This means that the templates can be developed in a certain degree of isolation, giving the developer the opportunity to concentrate on getting a single template right at a time. Therefore, even though many templates are created for only a single protocol, this is an efficient way to prototype protocols based on a CPN model.

Compared to previous examples, the WebSocket model had many more services. This makes the principal level somewhat harder to read and suggests that some kind of mechanism of grouping the services in several layers might be advantageous. At the service level, the models are approximately the same size as in previous examples. The readability of the service level modules can also be controlled by offloading behaviour to pragmatics such as we do for the $\langle\langle$receive$\rangle\rangle$ pragmatic in the message broker. All in all, the WebSocket model shows that we can make code generation models for real protocols without necessarily losing descriptiveness.

We have also showed that the code generation model can be verified by state space exploration. This highlights a major advantage of using CPN models which are directly executable. This allows us to perform analysis on high-level models and thereby keep the state spaces small. Although the verification presented only considers basic connection establishment and termination properties, other more elaborate properties including liveness properties can be checked using similar techniques. We are also working on using the sweep-line method, and advanced state space exploration method, to alleviate the state space explosion problem.

Finally, we have validated the automatically generated WebSocket implementation both by applying it to a well-known example in the form of the example chat application which is distributed with the GlassFish Application server and also by using the Autobahn test-suite which thoroughly tests most aspects of WebSocket protocol implementations.

To the best of our knowledge there does not exist any examples of using model-based techniques for generating an implementation of the WebSocket protocol in the literature. However, there exists a few examples for other industrial sized protocols. In [12] PP-CPNs, another class of Petri Nets, was used to generate code for the DYMO routing protocol for the Erlang platform. In our approach, we have a more flexible code generation approach through pragmatics that allows us to create new custom templates for new situations. Another approach to code generation is exemplified by the RENEW tool [13]. RENEW uses a simulation based approach where the implementation is a simulation of the underlying Petri Net. The direct use of simulation code makes it harder to meaningfully inspect the generated programs.

Other formalisms such the Specification and Description Language (SDL) has also been used as a starting point for code generation. For example, an early warning system for earthquake was developed using SDL in combination with UML [6, 3]. Both simulation and prototype code were generated using C++ as the target language. Our approach differs from the above mentioned approaches by the flexibility in abstraction level because of our pragmatics, by being platform independent by simply substituting templates and by the fact that we model the API explicitly at the service level and thereby easy interoperability with third-party software. Our approach also allows us to model the service interface, which is not available to the same degree in PP-CPNs and Renew. Another approach to generating software for reactive systems from UML models is the SPACE method [10] and its tool Arctis. This approach employs collaborations to compose services. The collaborations are then transformed into state machines that are executed together with Java snippets which are bound to actions of the collaborations. This approach relies on the state machines to either be translated to code or executed directly by some other tool. This is in contrast to our approach where we generate code directly instead of going through other formalisms and tools. MACE [9] is a textual state-transition language that is used to create distributes systems. It uses a compiler to compile the textual state-transition language into C++ code. This means that MACE is not as platform independent as our template-based approach is. In MetaEdit+ [19], models are mapped to some underlying formalism on which analysis is then performed. This tends to produce larger state space sizes compared with our approaches where the model is executable and allows verification at a high level of abstraction.

In the future we will apply more advanced state-space techniques, such as the sweep-line method, in order to do a more through verification of the model with larger and more complex configurations. Furthermore, we will investigate how errors in code generated by PetriCode can be traced back to the relevant pragmatics and model elements.

References

1. A. Gupta. *Chat Sever using WebSocket in GlassFish 4.* https://blogs.oracle.com/arungupta/entry/chat_sever_using_websocket_totd.
2. J. Billington, G.E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, 2004.
3. M. Brumbulli and J. Fischer. SDL Code Generation for Network Simulators. In *Proc of SAM '10*, volume 6598 of *LNCS*, pages 144–155. Springer, 2011.
4. CPN Tools. *Home Page.* http://cpntools.org/.
5. I. Fette and A. Melnikov. The websocket protocol. 2011. http://tools.ietf.org/html/rfc6455.
6. J. Fischer, F. Kühnlenz, K. Ahrens, and I. Eveslage. Model-based Development of Self-organizing Earthquake Early Warning Systems. In *Proceedings of MATHMOD*, 2009.
7. Groovy. *Project Web Site.* http://groovy.codehaus.org.
8. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.
9. C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN Notices*, volume 42, pages 179–188. ACM, 2007.
10. Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Compositional Service Engineering with Arctis. *Telektronikk*, 105(2009.1), 2009.
11. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Transactions on Petri Nets and Other Models of Concurrency VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
12. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
13. O. Kummer et al. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In *Proc. of ICATPN '04*, volume 3099 of *LNCS*, pages 484–493. Springer, 2004.
14. Oracle Corporation. *GlassFish Application Server.* https://glassfish.java.net/.
15. K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.
16. K.I.F. Simonsen. An Evaluation of Automated Code Generation with the PetriCode Approach. In *Submitted to: PNSE'14*.
17. K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. In *SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert*, volume 8368 of *LNCS*, pages 151–166. Springer, 2014.
18. Tavendo GmbH. *Autobahn—Testsuite.* http://autobahn.ws/testsuite/.
19. Juha-Pekka Tolvanen. Metaedit+: domain-specific modeling for full code generation demonstrated. In *proc. of OOPSLA'04*, pages 39–40. ACM, 2004.